Applicability of the session and the presentation layers for the support of high speed applications

Walid Dabbous, Christian Huitema, Leon Vidaller Siso, Joaquin Seoane, Julio Berrocal

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Rapports Techniques

N°144

Programme 1

Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

APPLICABILITY OF THE SESSION AND THE PRESENTATION LAYERS FOR THE SUPPORT OF HIGH SPEED APPLICATIONS

Walid DABBOUS
Christian HUITEMA
Leon VIDALLER SISO
Joaquin SEOANE
Julio BERROCAL

Institut National
de Recherche
en Informatique
et en Automatique

2004 route des Lucioles
B.P. 93
06902 Sophia-Antipolis
France

Octobre 1992

Applicability of the session and the presentation layers for the support of high speed applications.

Evaluation des couches session et présentation pour le support des applications haut débit.

Walid Dabbous, Christian Huitema

I.N.R.I.A. Route des Lucioles
B.P. 93
06902 Sophia-Antipolis
France

Leon Vidaller Siso, Joaquin Seoane, Julio Berrocal

Universidad Politécnica de Madrid
Ciudad universitaria
E-28040 Madrid
ESPAGNE

{dabbous,huitema}@sophia.inria.fr
{leon,joaquin,jerrocal}@dit.upm.es

Résumé Ce rapport résume les résultats d'une évaluation des services et protocoles de session et de presentation dans le but d'étudier leur convenance à supporter des applications à haut débit. Nous avons alors concentré sur les aspects concernant les performances de ces protocoles standards. L'étude est focalisé sur des problèmes tels que l'évaluation de la couche session, la convenance des langages de description de données, la transparence de données et les syntaxes de transfert et l'évaluation du protocole de présentation. Un intérêt particulier est porté à la description et à l'évaluation d'une syntaxe de transfert à coût léger appelée FTLWS (Flat Tree Light Weight Syntax) proposée comme alternative aux règles d'encodage BER de ASN.1

Abstract This report contains a detailed assessment of the session and presentation services and protocols in order to study their suitability to support high speed applications. We thus concentrate on the performance aspects of these standard protocols. The evaluation will be focused on the following topics: the aasessment of the session layer, the suitability of some Interface Description Languages, data transparency and transfer syntaxes, the presentation protocol. A special interest is given to the description and evaluation of a light weight transfer syntax called FTLWS (Flat Tree Light Weight Syntax) proposed as an alternative to the ASN.1 Basic Encoding Rules.

Chapter 1

Introduction

This report contains a detailed assessment of the session and presentation services and protocols in order to study their suitability to support high speed applications. We thus concentrate on the performance aspects of these standard protocols. The evaluation will be focused on four topics outlined in the next sections.

1.1 Assessment of the session layer

The assessment of the session layer can be done in terms of functionalities or in terms of performance. As the functionalities is studied in detail in task C2 (study of synchronisation functions), it was decided to focus this study on performance aspects and to distribute it along the following topics:

- Speed of coding: the session protocol uses its own coding rules, and thus cannot benefit from advances of the presentation technologies,
- Impact on upper layers performance: this is likely to derive from the complexity of the session protocol and from the "octet" orientation of the coding rules, which could jeopardize attempts to implement "word" oriented presentation syntaxes – that require alignment of data on "word" boundaries,
- Concatenation rules: the session protocol transmit the "last" SPDU first within a session message, which would not help the streamlining of the protocol.

1.2 Interface description languages

The current presentation protocol is based on the "Abstract Syntax Notation 1" (ASN.1). In order to investigate the suitability of this language for high speed multi-media applications, we will compare it to other similar languages:

- XDR, defined by SUN for its RPC system,
- NDR/NIDL, developed by Apollo Computer Inc, as part of their NCA, and selected by the OSF,
- IDL, the interface description language of the ANSA project.

The comparison will be however focused on the functionnal aspects of ASN.1 and XDR.

1.3 Data Transparency and Transfer Syntaxes

This part will discuss the performances of the ASN.1 "basic encoding rules" (BER) and of alternative transfer syntaxes like "FTLWS" proposed by INRIA.

As the performances depend from the quality of implementation, a first subsection (chapter 4) will be devoted to the analysis of the effort to optimize the BER, which was conducted by INRIA. The study of alternative encodings was also performed by INRIA. This chapter will also include a description of the implementation of the FTLWS syntax within the INRIA ASN.1 MAVROS compiler. We will then compare the gain of both the performance optimization and the alternative encodings in order to evaluate the necessity to use such light weight encodings for high speed applications. For this goal, we performed experiments with both BER and alternative encodings implemented within the INRIA ASN.1 compiler (MAVROS). We will also present comparisons of BER and XDR local tests (on the same machine) and over the network (Ethernet and FDDI).

The comparisons in chapter 5 shall address such points as the relative difficulty of optimizing a particular implementation (DIT), and the possibility to implement hardware support for particular implementations.

1.4 Presentation protocol

A number of critical points in the presentation protocol have been identified by past experience; their possible impact on high-speed applications will be reviewed. These points include:

- the cost of handling several encoding rules,
- the capacity to implement "fine grain" negotiation, e.g. selecting independantly a representation for integer and a representation of floating point numbers,

1.5 Conclusion

The chapter 7 will contain the general conclusions of this report.

Chapter 2

Assessment of the session layer

2.1 Introduction

2.1.1 Purpose

On the earlier 70's, when the OSI Reference Model was defined, the situation was of an emerging computer technology with more and more powerful machines and where the bandwidth of the communication lines was limited to a few tens of Kbits. The design of communication protocols, and OSI in particular, was focussed both on how to exploit the available line capacity and on problems derived from the poor quality of the transmission lines. But the CPU time was not considered as a critical factor.

Today the situation is quite different. Protocol designers and implementors must face the possibility of using reliable lines and networks at speeds over 100 Mbps. Therefore, the bottleneck is the hardware and communication software of the equipments and not the networks themselves.

In this paper we will address those aspects of the OSI Session Layer which could negatively affect its inclusion in a stack of protocols for high speed applications.

In the following, the service definition, the protocol design, the rules for encoding protocol data units and the concatenation mechanisms are reviewed.

2.1.2 Definitions

APDU: Application Protocol Data Unit

ASN.1: Abstract Syntax Notation One

MAP: Manufacturing Automation Protocol

QOS: Quality of Service

SPCI: Session Protocol Control Information

SPDU: Session Protocol Data Unit

SSDU: Session Service Data Unit

TOP: Technical and Office Protocols

TSDU Transport Service Data Unit

2.1.3 References

ISO 8326 Basic connection oriented session service definition.

ISO 8327 Basic connection oriented session protocol specification.

ISO 8072 Transport service definition.

ISO 8825 Specification of basic encoding rules for ASN.1

2.2 The session service

The function of the Session Layer in the OSI Reference Model is to "provide the means for organized and synchronized exchange of data between cooperating SS-users".

Through mechanisms such as selection of functional units and assignment of tokens the users have the possibility to tailor the service to meet the functional needs of their applications.

This is done by a three step negotiation during the connection establishment phase. The first one takes place between user and provider at local interface boundary, the second one is realized between the peer session protocol entities and the third one is the final negotiation between the end users.

Some of the application requirements are functional, being able to be satisfied by selecting the appropiated set of functional units, but others have special characteristics, being some of them closer to transport or network service features.

To deal with these latter requirements the session service defines a quality of session service.

2.2.1 Quality of Service

Two types of QOS parameters can be identified:

- Those that are directly mapped onto the corresponding transport QOS parameters. Throughput, transit delay and residual error rate belong to this group.
- Those that specify other special session service features.

The former group has a great importance but, since they are translated into their equivalent transport QOS parameter, their study is beyond the scope of this paper.

The latter group consists of parameters that can affect the behaviour of the session protocol:

- extended control, related to recovery mechanisms when normal flow is congested.
- optimized dialogue transfer, related to the concatenation mechanism of certain service primitives. Its influency is evaluated in section 2.5.

The selection of extended control allows the session service users to employ certain services primitives to interrupt a given connection when normal flow is congested. By their means, the users have an out of band mechanism to react against severe QOS degradations and decide how to recover resources when necessary. Basically the user has two possibilities, either abort the connection with probable loss of data or interrupt the current activity and resume it later when the conditions would be more favourable.

In a high speed environment, where it is sure that there would be applications with different bandwidth needings and with different data transfer priorities associated with them,

the coordination among sessions has special relevance. It is recognized the importance of determining how the communication resources can be dynamically shared among different applications. Perhaps the most significant resources that can be shared at session level are the transport connections. A transport connection may be reused, so producing an important improvement of the quality of session service at low cost, especially when application associations of short duration are considered. In these cases it is possible that the time needed to establish the transport connection would be comparable or even greater than the time employed in transferring the session user data.

2.2.2 SS-User Observable Throughput

Clearly, in a high speed environment, the major requirement to be satisfied by the session service is to allow high flow of information through its interface. In this sense, the session service definition does not impose any limitation on the amount of data that can be transferred. But it must be taken into account that the insertion of synchronization points and the use of activity management services can lead to the degradation of the throughput offered by the upper layers.

This is due to the number of service primitives that must be invoked and the associated protocol overhead and, fundamentally, for having to wait until the corresponding acknowlegment (synchronization) or confirmation (activity) generated at the peer user side arrives. To improve this value the rule is "the larger the SSDU size, the higher the throughput". But applications themselves or implementations may have to limit it.

2.3 The session protocol

The current service definition and protocol specification is the consequence of a consensus achieved during the development of the OSI session standards. The result is a protocol that combines features of the T.62 Teletex and the ECMA 75 protocols with some others mechanisms such as the tokens exchange.

ISO 8327 describes the protocol behaviour as an automaton with 29 states and 75 events that can fire transitions. Most of these events represent the invocation of a session primitive or the arrival of an SPDU, but also some of them are related with timer expirations.

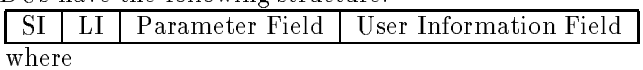
Fortunately the session protocol was designed to be modular, being possible to select a part (some functional units) of the whole protocol. For instance, MAP and TOP only support the full-duplex mode while NIST, which is less restrictive, explicitly only excludes the Negotiated Release and Capability Data Exchange functional units from the allowed subsets in its implementation guides.

The session protocol design cannot be considered complex. On the contrary, most actions are immediate, since a one-to-one relationship between service primitives and SPDUs exists. Therefore, the main work of the protocol consists in to serialize service primitives (encoding) and de-serialize SPDUs (decoding).

What can be inefficient are the implementations derived from the design, but this aspect will not be addressed in this document.

2.4 Structure and encoding of SPDUs

SPDUs have the following structure:



where

- The SI (Session Identifier) field consists of one byte that gives the SPDU type.

- The LI (Length Identifier) field is a value between 0 and 254 telling how many bytes of parameters field follow or, if there are more than 254 bytes, LI takes the value of 255, and two additional bytes follow it, giving the length. So the maximum length for one SPDU is limited to 64 kbytes.

- The parameter field, if present, consists of the PGI (Parameter Group Identifier) units and/or PI (Parameter Identifier) units defined for the SPDU.

- the user information field, if defined for the SPDU and if present.

Therefore, each SPDU and each of its parameters, single (PI) or structured (PGI), are encoded following a TLV (Type, Length, Value) format.

The complexity of the coding rules is increased due to the compatibility with the CCITT T.62 Recommendation. It forces to maintain some artificial rules, as for example that although one PGI with only one parameter is equivalent to one PI, they are encoded in different ways. So it is possible to have different representations for the same thing.

2.4.1 Assessment of the encoding rules

This kind of encoding is closer to the Basic Encoding Rules (BER) of ASN1. In fact, the session level was defined several years before that BER was developed, providing the basic idea for its definition. Consequently, they can be considered as the ASN.1 predecessor but they lack of all its advantages.

Some drawbacks are:

- There are no basic types. Each single parameter has its own coding rules that are different from others with similar definition:

  - The reason code in a REFUSE SPDU and in a EXCEPTION DATA that could be both considered like integer or enumerate types are encoded with rules slightly different.
  - The sequence numbers that are defined as integers in the range 0 up 999999 and where each digit is encoded in an octect.

- There are no structured types. Each PGI has its own and particular coding rules.

- Each PGI and PI, has a different identifier, so there is a large number of them.

Are the coding rules so unefficient that could become a barrier that endanger the usability of the session layer in high speed environments?

Clearly, time wasted in the coding process has effect on protocol performance, but its impact on it depends on the length and complexity of each SPDU.

A detailed analysis of the SPDUs structure reveals that the most complex ones are those related to the connection establishment phase. Thus, for example, the ACCEPT SPDU consists of one SPCI of 12 fields plus the user data. The user data are transparent for the session provider and only their lengths have to be taken into account to compose the SPDU. Their contents are simply appended, and so, they have a little influence on the time consumed. Obviously this would be true, if the local interfaces are well designed and the overhead produced to pass data between adjacent layers is minimized.

With this assumption, we can consider that only the SPCI structure by itself is the responsible of the time wasted.

But since the length of its longest field is 64 bytes and it is simply a sequence, we can conclude that although the rules and mechanisms associated with the coding process are

not the most adequate in terms of efficiency, their impact in the time consumed is not so relevant as compared with the time employed to handle APDUs.

Apart from this, what is clear is that it has no much sense to use a different syntax from those used in the upper layers. This situation is forcing to a progressive isolation of the session layer, which keeps it far from obtaining benefits from the continuous advent of new and faster presentation coding rules.

Regarding the implementation, it must be taken into account that, although the SPDU structure and related coding rules would not have a great influence on the protocol performance, there is no doubt that they have an important effect in the complexity and size of any resulting implementation. Implementing the session protocol leads to write a great number of lines for supporting the coding rules. The situation would be quite different if the protocol could use the same transfer syntax as presentation does, since then the same compilers could be used, saving in this way a lot of effort.

2.5 Concatenation rules

With the purpose of improving the performance and reducing the number of transport primitives that must be invoked, the session protocol allows to group several SPDUs into one T-DATA-REQUEST primitive.

Each SPDU is defined as belonging to one of three possible categories:

- Category 0 are those that may be concatenated or not.
- Category 1, are those that may not be concatenated.
- Category 2, are those that must be concatenated.

After one association between peer applications entities (AEs) has been established, APDUs are normally exchanged through the Presentation/Session Data Request service and then, the apdus are transparently transferred using the Data Transfer (DT) SPDUs. So, to evaluate the benefits of employing the concatenation rules it is necessary to consider how they affect the exchange of these SPDUs.

One DT SPDU consists of two fields, the enclosure item that indicates whether or not this SPDU is the beginning or end of SSDU and the user information field, which if present, shall contain user data supplied by the SS-users.

The user information field does not follow the TLV format. The complete DT SPDU is the result of appending the user data to the DT primary structure (SI + LI + enclosure item field) and when decoding, its length is determined from the length of the received TSDU. Because of that, it must always be the last field of a given TSDU and no more than one DT SPDU can be sent in the same TSDU. So one of the possible advantages of using concatenation is lost.

Besides DT SPDUs, which belong to the category 2, can never be sent alone. On the contrary they must be concatenated at least with another category 0 SPDU. In practice this means that each SPDU must include a prefix of two bytes corresponding to a null GIVE/PLEASE TOKENS SPDU.

Moreover, the rules do not permit that concatenated SPDUs be processed in the same order they are received. Instead, one precedence based criterion must be followed.

When the extended concatenation is selected, the category 2 SPDUs are processed before the category 0 SPDU and also the former SPDUs must be processed following a specific order that considers four precedence groups.

For example, if we consider the reception of the following sequence (ISO 8327, Table 8)

```
     (1)             (2)           (3)             (4)
GIVE TOKENS + ACTIVITY START + ACTIVITY END + DATA TRANSFER
```

they must be processed in the following order:

```
     (2)             (4)           (3)             (1)
ACTIVITY START + DATA TRANSFER + ACTIVITY END + GIVE TOKENS
```

So more proccessing is needed and the advantage of grouping several SPDUs in only one TSDU is partly lost.

2.5.1 Assessment of the concatenation rules

The concatenation rules have been designed to optimize the use of the transport service. Implementing them forces to use a temporary space (disk or memory) to store the encoded SPDUs and then to have an algorithm to decide when one TSDU has been completed and should be sent or in which order the received concatenated SPDUs should be given to the user.

Therefore, it is not clear whether it is better to implement such algorithms and reduce the number of transport service primitives to be invoked or not to use concatenation and reduce the code complexity.

What it is apparent is that no potential advantages are detected when this rules are applied to the DATA TRANSFER SPDU. Since this kind of SPDU neither can be sent alone or can be sent more than one of them in a given TSDU, the use of the concatenation rules contributes to a certain performance degradation.

This effect is more perceptible when the synchronize and activity management functional units are not selected and therefore only DT SPDUs are involved in the data transfer phase.

2.6 Conclusions

This chapter gives an overview of the OSI Session Layer Service and Protocol, stressing on the aspects than can be critical when high speed networks are considered.

In respect of the service definition, it is stated the importance of selecting carefully the most adequated values for the components of the quality of service parameter. Although the most of them are translated to their correspondent transport QOS, others like the extended control and the optimized dialogue transfer have influence on certain session features.

Also, it has been identified that the use of certain session services as synchronization or activity management can reduce considerably the throughput achieved by the upper layers.

Regarding the protocol specification, it is designed in a modular way with simple algorithms. An application requiring the whole functionality will probably include a large amount of code with trivial actions, and therefore inefficient implementations can be derived.

The main task of the protocol consists in to serialize service primitives (encoding) and to de-serialize SPDUs (decoding). The SPDUs structure is similar to BER of ASN.1, so the same rules could be adopted. Although current rules and mechanisms associated with the coding process are not the most adequate in terms of efficiency, their impact in the time consumed is not so relevant as compared with the time employed to handle APDUs.

In reference to the concatenation rules, a potential weakness is foreseen, although to reach a conclusion it is necessary to compare which is better, adopting concatenation rules and reduce the number of transport service primitives invoked or not to use concatenation rules and reduce the code complexity.

Chapter 3

Interface Description Languages

3.1 Introduction

3.1.1 Purpose

Assesment on external data description languages to check their suitability for High Speed and Multi Media applications. Although the assesment is general, it pays more attention to ASN.1 and XDR.

3.1.2 Definitions

ASN.1: ISO Abstract Syntax Notation One.

HS: High Speed applications.

MM: Multimedia applications.

DDL: Data definition language.

XDR: Sun eXternal Data Representation.

LOTOS: Language of Temporal Ordering Specification.

3.2 Concepts

Aplications exchange data among their components and the external world (let us think of it as another component). For these components to cooperate they need to know properties of the data they exchage. These properties are often expressed in the form of *data types* described in some formal *data type description language*. Despite not all languages used to develop applications have such a type definition language (i.e., LISP), it is widely accepted its need, specially for open distributed applications. Of course, it is even more necessary for applications to describe individual values with a *data value definition language*. Both languages jointly form a *data definition language* or *DDL*.

In order for open applications to cooperate it is necessary a specification on how they interact, that is, on the paradigm of interaction, on data exchanged in each interaction, and on temporal ordering on interactions. Paradigms are roughly classified in *message passing* and higher level, being the *remote procedure* the most popular. Multimedia applications

probably will require new high level paradigms, such as *streams*, *devices*, etc. High level paradigms usually need language support for the syntactic specification of interfaces of remote procedures, streams, devices, etc. This language is sometimes referred to as an *interface description language* and is closely related to the DDL. In this report we will not pay much attention to interface description languages in this sense, maily because remote procedure call is the subject of other task (C3).

3.3 Importance of data definition languages to HS and MM

The importance of DDLs to high speed applications is relatively marginal, because they are intended for specifiers and off line language tools. However their semantics strongly influences the kind of encoding rules for data, and even the structure of applications, so there is an unavoidable influence on speed. In fact, most of them were designed *after* the encoding rules.

The importance of DDLs on multimedia aplications comes from the need to comfortably define the data types needed for such applications, where in many cases there are pictures, sound, etc. Often these applications are well suited to some *object oriented* paradigms (for example, multimedia devices are easily mapped to objects), so it should be taken into consideration.

3.4 Classification of data definition languages

Data definition languages for external data may range from natural language specifications to algebraic data types. Let us briefly review some of them.

3.4.1 Syntactic text definitions

The simplest way to exchange data among distributed applications on heterogeneous machines is to use plain text in some agreed alphabet. These applications describe data as belonging to a context free grammar expressed in some syntax notation, such as Backus-Naur's form. For example, a `Date` can be expressed as:

```
Date ::= Day - Month - Year .
Day ::= Digit [Digit] .
Month ::= Digit [Digit] .
Year ::= Digit Digit Digit Digit .
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 .
```

Syntactic descriptions have been used for years in electronic text mail systems such as RFC822 [4]. Their main advantages are simplicity, human readability of messages, expressive power, and extensibility. They are also easy to process with code produced by hand or by automatic language tools such as *lex* and *yacc*, but bandwidth occupation and speed of processing are not astonishing. However, due to their high expressive power they have been extensively used to describe such things as documents, mathematical formulae and drawings. In such cases speed of decoding is usually irrelevant compared to the effort of processig the information.

3.4.2 Data structures

The next step is to define some basic types and define the others as the composition of the basic ones (integers, booleans, etc.). In this way the basic types are made abstract, so the application specifier is no longer concerned with their representation. However he is concerned with the representation of complex ones. Basic types can be represented in an efficient way (no longer human readable), and structures as their concatenation. This is the way conventional programming languages define types with utmost success with respect to efficiency/abstraction tradeoffs. This is also the way contemporary external DDLs, such as ASN.1 [8, 9] and XDR [13], define data too. For example, our `Date` may be defined in ASN.1 as:

```
Date ::= SEQUENCE { day INTEGER, month INTEGER, year INTEGER }
```

External DDLs have some general differences with internal ones, derived from the fact that data types in memory cannot have variable size. So external types can usually have variable size and even be recursive. On the other side pointers are not usually allowed, difficulting the definition of complex structures, such as graphs (the Internet Multimedia Mail System [12] is an exception to this rule; it supports pointers, called tags, but its data definition language resembles a syntax specification).

Structure based DDLs are often extended with constructs giving additional information on the values permitted for a given type, such as the maximum allowable range, values permitted, number of components, and the so on. This information is mainly used for resource allocation and, it is said, to offer a more precise specification and a tool for error checking. In our ASN.1 example:

```
Date ::= SEQUENCE {
   day INTEGER (1..31),
   month INTEGER (1..12),
   year INTEGER (1900..2900)
}
```

However syntactic constructs to express constraints make syntax more complicated and cannot be general enough to express simple constraints like *february has 28 days in normal years and 29 in leap years*. A predicate definition language would be more appropriate for such specifications.

3.4.3 Algebraic data types

We have seen that defining types as constrained data structures is powerful, but by no means general or abstract. The only abstraction is placed in the basic types. Algebraic specifications define data types by means of *sorts*, *functions* on them, and *equations*, giving the semantics. For example, a `date` can be expressed in LOTOS [10] as:

```
type Date is Integers and Booleans with
  sorts
     date
  opns
     epoch:                -> date
     cons:  int, int, int -> date
     day:             date -> int
     month:           date -> int
     year:            date -> int
```

```
    yesterday:      date -> date
    tomorrow:       date -> date
    daysof:         int  -> int
    ...
  eqns
    cons(1, 1, 1900) = epoch ;
    day(cons(d, m, y)) = d ;
    month(cons(d, m, y) = m ;
    year(cons(d, m, y)) = y ;
    yesterday(tomorrow(x)) = x ;
    daysof(1) = 31 ;
    (y mod 4) = 0) = true  => daysof(2) = 29 ;
    (y mod 4) = 0) = false => daysof(2) = 28 ;
    daysof(3) = 31 ;
    daysof(4) = 30 ;
    ...
    (d < daysof(m)) = true => tomorrow(cons(d, m, y)) = cons(d+1, m, y) ;
    ((d = daysof(m)) = true) and ((y < 12) = true)
                           => tomorrow(cons(d, m, y)) = cons(1, m+1, y) ;
    tomorrow(cons(31, 12, y)) = cons(1, 1, y+1) ;
    ...
  endtype
```

Semantics are useful for the specification of the behaviour of the protocol and for the implementation of types in the best concrete way for each partner in the communication. They provide directly or indirectly all necessary constraints on data (i.e. on the days of february on a leap year).

For communication purposes only the signature is absolutely necessary. A value may be sent in some form derived from the signature. For example, the sixth of January of 1900 may be sent in one of the following forms:

```
cons(6, 1, 1900)
tomorrow(tomorrow(tomorrow(tomorrow(tomorrow(epoch)))))
tomorrow^5(epoch)
```

Of course, for such a representation to be compact and the decoding efficient, a compact representation of some basic types is required. In the first representation here integer numbers are not derived from their equations, so it is easy to send a representation of 1900. For multimedia documents, mappings of sounds, images, etc. to bitstrings with efficient encodings should be defined, unless primitive representations are defined for them.

Using the data part of LOTOS as a data definition language is attractive because only one language needs to be used to wholly specify application services and protocols. Perhaps minor changes to the language are needed to express which functions are useable to convey information trough "the wire".

However the use of abstract data types may lead to ineficcient implementations, mainly because their functional nature and, perhaps, because of the difficulty to automate the encoding process based on concrete representations (that is, the implementor must define a *serializer* for each *sort* to be sent as abstract).

On the other side, the line representation may be extremely efficient for some frequent cases, specially if a rich set of constants and other functions is defined. In our example it is very economical to send the date `epoch`. This economy could be greatly improved if there is the possibility of dynamically extend data types with new functions defined as compositions

of older ones. This is exactly what is done by Sun's *News* protocol based in the *PostScript* [1] page definition language.

3.4.4 Objects

Objects are the non functional equivalent of abstract data types. They replace functions for *methods*, that is, procedures to see or change the status of an object, perhaps using other objects as parameters and giving new objects as results. Objects are usually specified as instantiations of classes, while methods are usually specified procedurally, in terms of instance variables, although some sort of abstract specification could be done in the form of predicates derived from the algebraic counterpart. Let us write only a syntactic interface of our example:

```
class Date is
  epoch(): Date;
  cons(int d, int m, int y): Date;
  day(): int;
  month(): int;
  year(): int;
  yesterday(): Date;
  tomorrow(): Date;
end Date
```

In this example methods returning a `Date` also change the status of the destination object to the same value, while *selectors* such as `day`, `month`, and `year` do not. (Returning the status is a trick to save code in object oriented applications, allowing one to write such things as `x.tomorrow().tomorrow()` instead of `x.tomorrow(); x.tomorrow()`).

Along with the concept of object oriented programming is almost always found the concept of *inheritance*. By means of inheritance one can define (sub)classes as enrichments or modifications of other classes, so that objects of inherited classes may have more information and more or modified operations.

There are two kinds of inheritance, namely *structural* and *functional*. With structural inheritance, subclasses may be defined extending the data structures used to implement objects. With functional inheritance (also called *conformance*), subclasses are just extensions of the interface (in the sense of [3], so details of implementations are hidden. For heterogeneous distributed applications seems that only functional inheritance is worthwhile, mainly because some objects may have very different implementations depending on the sites (i.e., cameras, speakers, etc.).

Another paradigm in object oriented design is *multiple inheritance*. Our interest in it is because it allows us to build composite objects as the sum of two or more interfaces. For example, an audio input/output system can be seen as a speaker plus a microphone.

Object oriented systems are either typed or untyped. Smalltalk is untyped, while C++ or Eiffel are typed. It is widely accepted that typed systems are better for efficiency and security, but they are often too inflexible. Type systems need the capability to define parametric classes (often called generic), that is, classes dependent on one or several types having certain properties. For example, it may be worthwhile to define a class `file of T`, able to be instantiated as a `file of integer` or as a `file of Date`.

Object orientation have become a buzzword in programming methodology of this decade, despite of the controversy on it. Many distributed systems claim to be object oriented in the same way many other systems claim. However many of these systems only handle objects and classes in a limited way. This is mainly due to the difficulty to implement some of the object oriented paradigms in a distributed system.

In these systems, objects are usually created in a server owning the appropriate class, staying there until they are destroyed. These *heavy* objects are referred to through references (often network wide *capabilities*) that can be passed among partners. Operations on remote objects seldom pass objects as parameters, but ordinary data structures, so that network traffic due to object references is kept to a minimum.

Of course, there are systems where objects are movable ([2, 5]), requiring to express the implementation of movable objects as data structures, or a way to serialize/deserialize them.

Linguistic suport for the definition of distributed object oriented systems require a way to define interfaces from scratch or using inheritance. Support for generic classes is also necessary. One example can be seen in [6]. Highly interactive applications may require to define objects able to interface with continous streams (voice, video), so linguistic support is needed for them. A construct is defined in [11].

Because the close relationship of object oriented interfaces and remote procedure calls, a detailed discussion on related languages is left for future refinement.

3.5 Detailed assesment on ASN.1 and XDR

As widely known data definition languages based on data structures, ASN.1 and XDR will be compared from a general point of view, taking into account their relative merits for high speed and multimedia.

Both languages are apparently very similar, although ASN.1 is exceedingly more complicated. Both have been designed after their encoding rules, so they mirror them in a way or another.

In the following we will make some critical judgements specific for each language, then a judgement on problems common to both languages for complex applications like multimedia. Some knowledge on both languages is assumed.

3.5.1 Specific problems of ASN.1

ASN.1 was designed for a complicated application (X.400 mail). Messages were complex nested structures with many optional parts. Their size could be huge, so conceptual support for unordered data was introduced. Generality was another goal, placing no limits to the size numbers and allowing a wide variety of character sets for strings. Last, but not least, the whole language was designed for extensibility of protocols, so existing implementations of protocols could interwork (in a limited way) with new enhanced standards.

The main specific problems with ASN.1 are not directly related with high speed or multimedia. Being a data structure definition language shares the virtues and problems of them. However it has some specific drawbacks:

1. The *selection* type is superfluous.
2. Some constructs such as *COMPONENTS OF* and *IMPLICIT* or *EXPLICIT* specification of tagging are only notational noise intended to reduce the bulk of the encodings.
3. Four types of tags are unnecessary (at most *UNIVERSAL* and *context* specific are enough for everything).
4. Even explicit mention of tags is noisy. Perhaps they should be generated automatically if the underlying encoding requires it. The only problem (if any) with this is that the order of writing is significant (the same applies to *ENUMERATED*).

5. Explicit mention of tags implies their inclusion in the encodings, making them inherently slower than languages without tags (i.e., XDR).

6. Perhaps the distinction between SET and SEQUENCE is not necessary, being a big problem for many decoders.

7. *Subtyping* is too complex and not general enough. Probably it is only worth to suggest resource limits in implementations. If more information is wanted, an equational language may be better.

8. *Tagged* types make no sense and raise questions like: is a tagged type a subtype?

9. There is not syntactic support for *abstract syntax*. In my opinion some *modules* should export one abstract syntax (a CHOICE type) with the same object identifier of the module.

10. *Macros* are a brain damaging, and difficult to implement. INRIA has now a version of the ASN.1 compiler which supports partially the automatic Macros generation.

11. Parsing is difficult (infinite lookahead needed), exceedingly difficult for macros.

12. Character sets perhaps should be supported as a presentation matter, specially those are not to be parsed, only repeated or printed. In fact, some data types (i.e., *UTC-Time*) are represented as strings and, probably, should not.

3.5.2 Specific problems of XDR

XDR was designed for simple and efficient applications in mind, usually implementable as remote procedure calls on top of datagrams. It is mainly an extension of Xerox Courier syntax [15], adapted to newer 32 bit machines and the syntax of C. The basic types and structures are defined after their encodings, so the definition of new encodings means the invention of new type names (for example, there is an *hyper integer* for a 64 bit integer; from the standard is suggested the possibility to define a *little endian integer*). The main problems specific of XDR are related to its simplicity. Let us enumerate some of them:

1. Representation negotiation and extensibility are implicitly banned because of the direct relationship of types and encodings (is this really a problem?). However, minor reworking may lead to negotiable and extensible representations.

2. Complex protocols make heavy use of optional fields in structures. This is only supported in XDR through *discriminated unions*, but their use for this purpose is awful.

3. There is no modularity.

4. There are not *bit strings*.

3.5.3 Common problems of ASN.1 and XDR

Both ASN.1 and XDR share some problems for the more sophisticated applications, some of them could be multimedia. Let us enumerate them in an arbitrary order of importance:

1. There are no generic (parametric) types. Parametric types may make some specifications shorter without increasing language complexity too much (and without using macros or ANY). For example:

```
Tree[T] ::= SEQUENCE {
  node     T,
  branches SEQUENCE OF Tree[T]
}

TreeOfIntegers ::= Tree[INTEGER]
TreeOfREAL ::= Tree[REAL]
TreeOfTreesOfIntegers ::= Tree[TreeOfInteger]
```

Parametrization may allow to define parametrised *application service elements* so that *real applications* instantiate them to the apropriate types before attempting a presentation connection.

2. There are no pointers for complex data structures. With them, complex data structures, such as `Graph`s could be represented. However *pointed to* data must be marked to be remembered in the decoding process until the end of the datum:

```
Graph ::= SEQUENCE {
   node Node POINTED,
   arcs SEQUENCE OF POINTER TO Node
}
```

However it is better to send nodes the first time they are referenced, making it easier the serialization and deserialization processes:

```
Graph ::= SEQUENCE {
   node Node ONCE,
   arcs SEQUENCE OF Node ONCE
}
```

Complex data can be represented within the framework of ASN.1 and XDR, but some tricks must be employed:

```
Graph ::= SEQUENCE {
   node Node,
   arcs SEQUENCE OF INTEGER  -- Number of node
}
```

Here the pointers are represented as integers, but comments are needed to clarify the meaning. This means too that automatic serialization and deserialization is impossible.

3. ASN.1 and XDR lack of the concept of *type extensibility*. This concept comes from object oriented design and was introduced by Niklaus Wirth [14] to permit software extensibility in the simplest way. In this way a new enhanced standard could be defined as an extension of an older one, that is without rewriting it (writing only the differencies). On the other side, some protocol may want to work on different levels of abstracion for the same things. The typical example is for graphical applications; we may want to define several kinds of `Figure`s and be able to send them in a message without the need to enumerate all possible cases in a CHOICE (perhaps they are not known in advance):

```
Fig ::= SEQUENCE { center Point, angle REAL }
Figure ::= SEQUENCE Fig { attr SEQUENCE OF Attribute }
Square ::= SEQUENCE Figure { side REAL }
Circle ::= SEQUENCE Figure { radius REAL }
Ellipse ::= SEQUENCE Figure { major REAL, minor REAL }
ComplexFigure ::= SEQUENCE Figure { parts SEQUENCE OF Figure }
GraphicWindow ::= SEQUENCE {
   corner Point,
   heigth REAL, length REAL,
   title TEXT,
   contents Figure
}
```

Implementation of type extensions require automatic or explicit tagging of extra (extended) components and will render CHOICEs unnecessary (in fact they can be implemented like them). Of course type extensions are just a notational convenience to implement extensibility and can be circumvented with current ASN.1 with self delimiting structures and a weak type checking in the decoder (note that the use of the *selection type* resembles the purpose of type extensions).

4. Some protocols are carriers of data, possibly relaying it through several machines (i.e., mail). In this cases portions of data types must be of any type and any encoding rule. So something is needed to represent unknown types. ASN.1 uses the *EXTERNAL* type to escape to any known *presentation context*, while XDR uses the *opaque* data type to encapsulate external encodings known in advance or encoded in a previous field. The problem with ASN.1 and EXTERNAL is with the presentation protocol, unable to negotiate types and rules for mail, for example.

5. Sometimes the data description language is not appropriate for some data item: text, voice, music, video, animation are to be repeated or sent to some hardware/software device. So an escape is needed to the different paradigm. Again ASN.1 uses the *EXTERNAL* type and XDR the *opaque* type for this purpose. In principle one may think that some keywords for them are approriate (i.e., TEXT, VOICE, VIDEO, etc.), but this is against the principle of stability of standards.

3.6 Conclusions

This work is just a blueprint with consideration of several alternatives of data definition languages. Perhaps some or all of them can be merged to get the best of them, but this is not probable. Only experimentation with different kinds of realistic applications will lead to the better designs for such groups. In the next chapter we will present the results of the tests accomplished with both ASN.1 and XDR in order to measure the suitability of each one of the two languages to support high speed applications.

Chapter 4

Data transparency and Transfer syntaxes

4.1 Introduction

This chapter studies the data presentation function and its impact on the preformance. We will compare both ASN.1 BER and XDR syntaxes. The ASN.1 Basic Encoding Rules will be assessed in section 4.3 and an alternative encoding for ASN.1 named FTLWS (Flat Tree Light Weight Syntax) will be described in section 4.4. This transfer syntax is now supported by MAVROS, the INRIA ASN.1 compiler (see 4.3.1). In section 4.5, we present and analyse the results of the experiments on both BER and FTLWS.

XDR was briefly introduced in chapter 3. In section 4.6 we will present the results of the performance tests with XDR encoding and decoding functions. We also performed some experiments (coding-transfer-decoding) on the INRIA local networks (Ethernet and FDDI) and will present in section4.7 the comparison of the acheivebale throughput for both ASN.1 BER and XDR. Finally, we will conclude about the suitability of these encoding rules for high speed applications.

4.2 Data presentation function

Computer networks link together various kinds of computers. Diversity, however, has its cost: applications can no longer communicate by copying fragments of data from their memory to the network, or to files: the structure of the data would be lost, as the binary representation of the same high level construct will vary in different systems. Binary incompatibility derives from three major causes: different *hardware*, different *programming languages*, or different *compilers* of the same languages.

The networking software has to adapt by providing what the OSI reference model calls a *presentation* service. The presentation function will encode the data from the local format into some *common* or *external* form before transmission, and will decode the received data from the common form to the local format after reception. Different computer manufacturers have proposed different protocols to perform this task (like Xerox's Courrier [16], Sun's XDR [17], HP's NIDL[18]) Partridge and Rose compared in [19] the functionalities of these external data formats.

Within the *Open System Interconnection* framework, ISO and CCITT standardized a presentation service [21], a presentation protocol [20], a data structure definition language called *Abstract Syntax Notation 1* (ASN.1)[22] and an associated data transfer syntax [23],

the *ASN.1 Basic Encoding Rules*. Chapter 3 is concerned with a detailed functional comparaison of ASN.1 with other interface description languages. In this chapter we will focus on performance aspects and will present the results of benchmark tests for both ASN.1 and XDR.

In the next section, we will analyze ASN.1 and its basic encoding rules, and show that they are extremely CPU demanding. Then, we will explain how one could devise alternative encodings rules and use them for faster data transmission.

4.3 The ASN.1 Basic Encoding Rules

The ASN.1 language is an upward compatible evolution of the CCITT recommendation X.409 [24], which was initially designed to provide a data transfer syntax and a data notation language for message handling systems. Unlike X.409, it clearly separates data specification, i.e. the definition of an *abstract syntax*, from data encoding according to a *transfer syntax*. The abstract syntax of the data elements is described in ASN.1 *modules*, using basic types and construction rules. The basic types are the BOOLEAN, INTEGER, ENUMERATED and REAL types, plus the BIT STRINGs and OCTET STRINGs and a special type called OBJECT IDENTIFIER, which provides unique indexes for various types of entities and protocols, coded on a small number of octets.

As most programming languages, ASN.1 includes facilities for building complex objects by combining these basic elements in structures, arrays or alternatives CHOICEs. Unlike other programming languages, ASN.1 includes two different notations for structures: the elements of a SEQUENCE must always be transmitted in their order of declaration, whilst the elements of a SET can be encoded in arbitrary orders. Similarly, the order of the elements is significant in a SEQUENCE OF, but not in a SET. A tagging mechanism allows for the distinguishing of different occurences of the same type having different semantics (i.e. when building new data types from old). This results in a unique representation of the type within a given module (APPLICATION class tags) or a given context (CONTEXT class tags), or according to a bilateral agreement (PRIVATE class tags). The basic types have a globally unique tag (UNIVERSAL class).

Is is the possible to specify that some components of a construct are OPTIONAL, or that they can take a DEFAULT value. The language also includes a NULL type, which can be used to encode an empty value, and the construct ANY, which can be used for inserting "opaque" data, i.e. that can take any legal ASN.1 value.

According to the ASN.1 "Basic Encoding Rules"[23], data elements are encoded with a "T-L-V" recursive scheme where "T" represents the type of data field, "L" its length, and "V" the actual data value. In addition, almost all fields have a variable-size encoding, with a requirement to use exactly the minimum number of bytes necessary to convey the value. This enables flexibility, in the sense that there is no limits in data size: for example, ASN.1 supports arbitrarily wide integers.

The systematic use of type and length field introduces unnecessary overhead on the network, since in many cases the receiver knows what data are being sent to it. In addition, the conversion process is highly CPU consuming because the ASN.1 encoding rules are based on bytes and variable-size representation:

- The elements of type INTEGER should be encoded using the exactly sufficient amount of octets. For example the value 127 would be encoded on three bytes ($T = 2, L = 1, V = 127$), but 128 would be encoded on four bytes ($T = 2, L = 2, V_1 = 0, V_2 = 128$).
- The encoding of the Length fields in the triplets has the same complexity as that of INTEGERs in order to use minimun number of bytes to encode the length field: if the length is lower than 128, a single byte is used; otherwise, a first byte indicates

transformation	*SPS7*	*GOULD*	*Sun 3*
assign (in a loop)	3.7	2.0	2.9
memory copying (per set of 256 int items)	0.9	2.9	1.1
encoding using ASN.1 transfer syntax (function)	18.6	10.9	25.5
in line coding using ASN.1 transfer syntax	10.9	6.8	12.5
decoding using ASN.1 transfer syntax (function)	20.9	10.7	24.3
in line decoding using ASN.1 transfer syntax	12.2	6.2	15.2
swapping from big endian to little endian	7.0	4.6	8.1

Figure 4.1: CPU time necessary to transform a 32 bit integer value (μs.).

the length of the length field, and the following bytes encode the actual length of the value. This encoding can be escaped, in the case of structured components, by coding the length as "indefinite", and then terminating the content by an "end of content" mark.

- The elements of type REAL can be encoded via several different formats, using either an ASCII representation or a binary representation. In the latter case, both mantissa and exponents can be encoded on a variable number of bytes, and the exponent can be expressed as a power of either 2, 8 or 16.
- When decoding arrays, it is only after decoding all the components that one can determine their number: memory allocation is not made any simpler by this choice.

These choices were made in order to have a general specification language and encoding rules. One can argue that ASN.1 is too general resulting in a low efficiency of its Basic Encoding Rules. In fact, we have observed a ratio of 5 up to 20 between simple memory copy, and conversions using the ASN.1 transfer syntax (see figure 4.1): the efficiency of the ASN.1 "basic encoding rules" is clearly debatable.

4.3.1 Serializing data elements

The MAVROS environment allows to build automatically a *C* programming interface to any "package" given its ASN.1 definition. This environement is centered around a couple of tools (MAVCOD and MAVROS) and a run time support (the ASN.1 library). MAVCOD acts as a preprocessor for MAVROS in the sense that it reads the ASN.1 specification and produces a set of *C* structures which correspond to the ASN.1 types in the specification and an augmented ASN.1 specification that contains the directives necessary to tell MAVROS how to map the various data objects. MAVROS produces an encoder which takes machine-dependent data types and produces an external representation of the data; and a decoder which performs the inverse operation. The "cost" of data processing (coding/decoding) forms the main overhead of the presentation layer.

Hopefully, there is at least one merit in the OSI presentation protocol: it incorporates a negotiation phase. An application (entity service) may use a set of one or several abstract syntaxes (e.g. ACSE and FTAM syntaxes) in order to describe all the data types that will be manipulated on a given connection. For each these abstract syntaxes, several transfer syntaxes, i.e. encoding rules, can be proposed by the calling presentation entity; the responding entity will choose the most appropriate syntax; then, the chosen transfer syntax will be used. Indeed, most implementations have a very conservative negotiation procedure by supporting only the ASN.1 Basic Encoding Rules. But nothing forbids us to realize more intelligent implementations, e.g. by proposing:

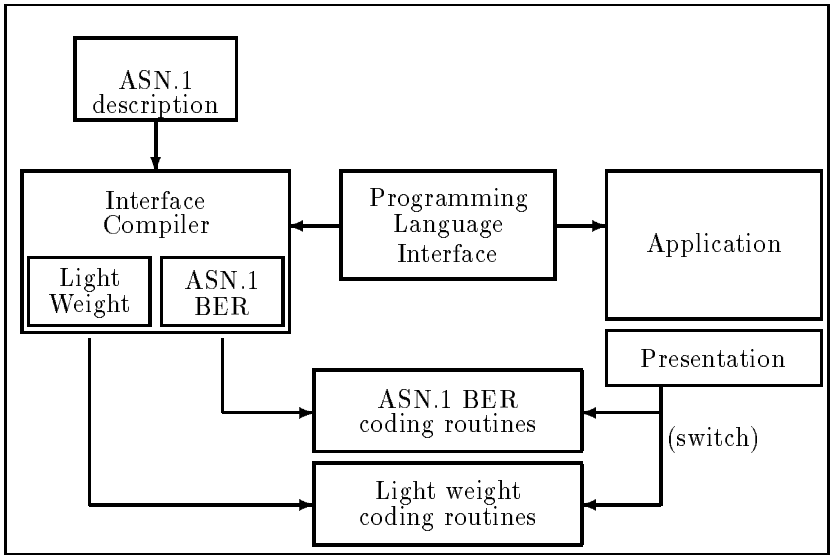


Figure 4.2: Compiling alternative transfer syntaxes.

- the ASN.1 Basic Encoding Rules, in order to guarantee interoperability with straight jacket implementations,
- a compressed encoding, for use on low speed networks,
- light weight encoding rules, faster to process than ASN.1, for high speed transactions.

Within the MAVROS environement, introducing a new "transfer syntax" is equivalent to changing the *code generation* part of the interface compiler, in order to generate not only the ASN1.BER encoding and decoding routines, but also the "light weight" encoding routines based on the new syntax. The choice of using either set of routines will be performed by the "presentation entity" on a connection per connection basis, depending on the result of the presentation negotiation. This will be fully transparent to the application, which will always uses the same "local syntax", defined by a programming language interface (figure 4.2). In the next section, we detail the light weight encoding rules and compare them to the BER.

4.4 Light weight syntaxes

INRIA developped a family of light weight syntaxes called FTLWS (Flat Tree Light Weight Syntax). Its specification was based on three principles:

- avoid unnecessary information in the encoding,
- use fixed representation when it is possible and,
- simplify the mapping of the elements by fixed length structures.

The first principle leads to the abandonment of the systematic "Type-Length-Value" encoding, which is replaced by new encoding rules for the ASN.1 constructs; the second and the third principles lead to a set of easily decoded *word oriented* encoding rules. The *word size*, is a parameter that characterize the syntax (16, 32, or 64 bits). The encoding of a

data value is defined a sequence of *words*, *pointers* and *strings*. A word is a unit of encoding defined by a word length and a bit order (big endian or little endian). A variable length element is accessed through a *pointer*[1] that specify a positive displacement between the current location in the encoding buffer and the encoding of the variable length element to which it "points". Strings encoding always begin at a word boundary. The encoding of the basic ASN.1 types is as follows:

- *BOOLEAN* elements are encoded on a single word.
- *INTEGER* elements are encoded on a single word.
- *ENUMERATED* elements are encoded as INTEGERs.
- *REAL* elements are encoded as indicated in the IEEE standard format for floating point numbers [27], on 8 octets containing a "double sized" number.
- *BIT STRING* elements shall be encoded as:
  1. A length, expressed as a number of bits, coded on a single word,
  2. A pointer to the string, coded on the word immediately following the length,
  3. A string, coded further in the message.
- *OCTET STRING* elements shall be encoded as BIT STRINGs with the length field indicating the number of bytes.
- *NULL* elements are not encoded.
- *SEQUENCE* elements shall be encoded as a length field followed by the encoding of each of non optional types, or of a pointer for each of the optional types. The length field is set to the number of words necessary to encode the mandatory components, plus one word per optional component. This length is fixed for a given specification of an abstract syntax, but may be augmented if the syntax is "extended".
- *SET* The encoding of a set value shall be the same as that of a sequence value of equivalent definition.

The complete definition of the FTLWS encoding rules is given in [28]. The merit of "pointer" approach is to use simple binary copy routines to serialize the values in the transmission buffer, copying first the fixed size elements and then the content of the variable size strings. The complexity of the decoding is similar: apart from a small number of pointer computations, we have traded element by element copying for and optimized binary copying procedure.

4.5 Performance Measurements: BER vs FTLWS

We have done some experiments to compare the performance of both ASN.1 (BER and FTLWS) and XDR. We also derives some possible implementation enhancement to the MAVROS compiler in order to speed up the data presentation function.

[1] Pointers have the same length as words.

4.5.1 Data Types and Values

The tests were performed using both basic and complex data types. The ASN.1 specification of the "object" of the performance test is the following:

```
perf-object ::= OPAQUE CHOICE {
        mix[0] SET OF SEQUENCE {
                a INTEGER,
                b BOOLEAN,
                c PrintableString},
        ints[1] SEQUENCE OF INTEGER,
        strings[2] SEQUENCE OF PrintableString,
        real[3] SEQUENCE OF REAL,
        mpdus[4] SET OF MPDU}
```

The test object is either a basic data type such as SEQUENCE OF INTEGER, SEQUENCE OF REAL, SEQUENCE OF PrintableString or a mix of some of these basic types. In order to test the encoding rules with more complex types, we chose the MPDU format (Message Protocol Data Unit) which defines the envelope of a message according to the 1984 version of the P1 Protocol (CCITT Recommendation X.411). The structure of the envelope is given in Appendix A.

The description of the data values in the tests is given in table 4.1.

The MPDUs values were obtained from ASN.1 encoded X.400 messages in order to make the tests with "real application" data.

4.5.2 The test functions

For every data type, MAVROS will generate the *C* procedures for the coding and decoding functions. We will concentrate on four groups of functions corresponding to each of the following syntaxes:

- ASN.1 BER: the basic coding and decoding functions according to the Basic Encoding Rules (`cod` and `dec`), plus a length test routine (`len`) that will return an overevaluation of the size of the ASN.1 encoding, and a copying routine (`cpy`) which is equivalent to calling `cod+dec`.
- ASN.1 "D"ER: the coding and decoding functions according to the "Distinguished Encoding Rules" (`xcod` and `xdec`) which requires that the defined format of the length field is always used and that the components of a SET are sorted by increasing tag order.
- ASN.1 FTLWS: The coding and decoding functions for the light weight syntax (`ftcd` and `ftdc`) plus the length overevaluation function (`ftln`).

Test identifier	Data Type	Number of test values
`ints`	SEQUENCE OF INTEGER	400
`real`	SEQUENCE OF REAL	200
`strings`	SEQUENCE OF PrintableString	124
`mix`	SET of SEQUENCE	36
`mpdus`	SET OF MPDU	18

Table 4.1: Data values used in the performance test

- Textual: the text coding and decoding routines that will either produce a pretty print in ASN.1 format of the parameters (`out`) or parse a text encoded area and set the result in the argument list which has the same format as that of the ASN.1 decoding routines (`in`), plus the the length overevaluation routine (`olen`).

These routines (`cod`, `dec`, `len`, `cpy`, `xcod`, `xdec`, `ftcd`, `ftdc`, `ftln`, `out`, `in`, `olen`) will be repeated 100 iterations in a loop and the average time per call will be calculated. We also produce an execution profile of the program in order to tally the number of calls to each routine and the number of millisecond per call.

4.5.3 Performance tests

Were performed several series of tests on a SUN 3/60 workstation. The results will help to find which possible enhancements are to be made to the MAVROS compiler. For all the tests, we computed the time and memory space used while coding of decoding the data. As we are interested in optimizing the coding rules for high speed transmission, we will focus on the completion time of the routines and disregard memory space considerations.

Test 1

The results of the first series of tests are given in tables 4.2 and 4.3.

We can make the following observations:

- The FTLWS coding routines are 1.6 to 5.8 times faster than those of the BER according to the data types. For basic data types, the improvement is considerable (specially for the `real` type) because of the relative efficiency of the word oriented coding.
- DER is always slower than BER because of more frequent sorting and test operations.
- FTLWS still more efficient than BER for the `mix` and `mpdus` types, but its advantage is reduced.
- Decoding was always more costly than coding (mainly because of memory allocation).

Test 2

After the first series of tests, the MAVCOD compiler was revised, in order to optimize the handling of OPTIONAL Strings. This should mostly make a difference for the `mpdus` test. The results are given in table 4.4 on page 26.

The result of this first optimization is not exactly conclusive. In fact, when we looked closely at some MPDU values, we observed that the elements which are designed as "optional strings" were seldom used. But the default test that we used (x.l=0 && x.v=0) is twice longer than the default test of the previous version (x = 0). As a result, the decoding took a little longer:

Coding routines	`ints`	`reals`	`strings`	`mix`	`mpdus`
BER (`cod`)	5999	13666	7833	3999	62330
DER (`xcod`)	12499	20999	17165	35498	249490
FTLWS (`ftcd`)	1333	2333	5499	2666	37998
Text (`out`)	69497	228824	46998	18665	221157

Table 4.2: Test 1: coding time in μs for different types and syntaxes.

Decoding routines	ints	reals	strings	mix	mpdus
BER (dec)	24665	302321	17499	8999	291821
DER (xdec)	26165	302154	20999	9666	283821
FTLWS (ftdc)	3333	3166	7666	4333	243490
Text (in)	58164	217491	121328	52164	2480234

Table 4.3: Test 1: decoding time in μs for different types and syntaxes.

routine	mix	mpdus
cod	4333	55497
xcod	35498	249490
ftcd	2499	40498
out	18499	222157
dec	8999	307154
xdec	9499	295154
ftdc	4499	247656
in	52664	2576230

Table 4.4: Test 2: with better handling of OPTIONAL Strings.

routine	Test 2	Test 1
dec	307154	291821
xdec	295154	283821
ftdc	247656	243490
in	2576230	2480234

This "optimisation" should then be left optional.

Test 3

By looking further, we observe that in fact there are a number of frequently used optional fields of the type:

```
SEQUENCE {
....,
[5] XXXXX OPTIONAL,
....}

XXXXX ::= PrintableString
```

If we had been able to "reduce" these references, then we could have also reduced the usage of "malloc" and "free", and perhaps observed an amelioration.

Following the previous pass (Test 2), the compiler was modified again, in order to implement the "type reduction" technique suggested above. We do observe some progression of the performances, from 10 to 12 %, depending of the syntax we consider:

routine	time in μs
`cpy`	340819
`len`	13999
`cod`	51997
`dec`	269989
`xcod`	239157
`xdec`	263822
`ftln`	12166
`ftcd`	35998
`ftdc`	221157
`olen`	19165
`out`	220824
`in`	2539231

Table 4.5: Test 3: with type reduction (`mpdus`).

routine	Test 3	Test 2	Test 1
cpy	340819	359818	386817
dec	269989	307154	291821
xdec	263822	295154	283821
ftdc	221157	247656	243490
in	2539231	2576230	2480234

Similarly, we could gain a lot (zero cost) by implementing static choices wherever this was possible.

Test 3 bis

In this pass of optimization, memory allocations have been removed as much as possible. Also, we have tried to remove all "repeated indirections", replacing them as much as possible by direct references to the data type. Finally, an optimisation pass has been installed, that removes all "unecessary" modules from the final code. The size of the source code for the whole module was more than 500K prior to optimisations; it has dropped to less than 360K.

The performance gain is noticeable:

routine	Test 3 bis	Test 3	Test 2	Test 1
cpy	264989	340819	359818	386817
dec	187325	269989	307154	291821
xdec	189159	263822	295154	283821
ftdc	144827	221157	247656	243490
in	1481440	2539231	2576230	2480234

The new version is 34.5 % faster than the previous one for the "light weight" syntax, 30.6 % faster for the "BER" syntax, 41.6 % faster for the "text" syntax, and 22.2 % faster for the "copy" procedure. If we do a comparison with the initial version, we observe a gain of 35.8 % for the "BER" syntax, 40.6 % for the "light weight" syntax.

Apart from the text syntax, which is not exactly aiming at performances, we observe that the ratio of cost from decoding to coding varies from 3.5 to 4:

$$\texttt{dec} \,/\, \texttt{cod} \quad = \quad 187325 \,/\, 51664 \quad = \quad 3.6$$
$$\texttt{ftdc} \,/\, \texttt{ftcd} \quad = \quad 144827 \,/\, 35998 \quad = \quad 4.0$$

routine	time in μs
`cpy`	264989
`len`	13332
`cod`	51664
`dec`	187325
`xcod`	203491
`xdec`	189159
`ftln`	11499
`ftcd`	35998
`ftdc`	144827
`olen`	18832
`out`	233823
`in`	1481440

Table 4.6: Test 3 bis: complete type reduction + module optimization (for the `mpdus` type).

There may well be some room left for optimizations, in particular:

- a better memory management,
- a version with security checks removed,
- a globalisation of security checks within the "light weight" routine,
- an implementation of the "mappings" authorized for the FTLWS routine,

These optimisations will be done in the next version, and will be incorporated to the INRIA-2 deliverable.

Test 4

As we observed in the previous tests, the decoding for both "BER" and FTLWS is more costly than coding. It is mainly due to memory allocation routines. In fact, all tests of the decoding routine, by default, are followed by a disallocation of the memory allocated during the decoding. A special test was conducted to mesure the relative costs of the decoding itself and the "free" routine. The first test was done with "mix"; the second one was done with "mpdus". The test was not very conclusive in the case of the "mpdus", as the process size quickly ran over 17 Megabytes, amid intense swapping and a near collapse of the system.

The mesurements were repeated in Test 4 bis, where each routine was tested in isolation for the "No free" test; this did not improve the case of the copying routines, for which we had to lower the number of test from 100 to 10 in order to obtain significant results (Test 4 ter). Note that this results also showed a noticeable improvment for the light weight routines.

We can see that the memory handling has a cost of the same order of magnitude as the "coding" routines. There is no reason to observe a difference between the cost of disallocation for the BER and light weight processes; the difference observed here should be taken as an indication of the precision of the mesurements. The cost of disallocation appears to represent about 1/2 of the cost of decoding, or about the order of magnitude of the cost of leight weight decoding.

If we take off the cost of memory handling, we observe that the best case light weight decoding is about twice as fast as the best case BER decoding for the complex MPDU

routine	decoding time	decoding without "free"
cpy	14166	6499
len	499	499
cod	4666	4333
dec	8666	7166
xcod	35831	36165
xdec	9166	7999
ftln	666	666
ftcd	2333	2333
ftdc	4166	2666
olen	1166	1166
out	18332	18165
in	57664	54831

Table 4.7: Test 4: Mesuring the cost of memory disallocation ("free") for the `mix` type

type	routine	test	Std decoding	No free	Difference	Coding
`mix`	`cpy`	4	14166	6499	7667	
`mix`	`dec`	4	8666	7166	1500	4333
`mix`	`ftdc`	4	4166	2666	1500	2333
`mpdu`	`cpy`	4 bis	264989	235990	28999	
`mpdu`	`cpy`	4 ter	261656	121661	139995	
`mpdu`	`dec`	4 bis	187325	138661	48664	51664
`mpdu`	`dec`	4 ter	194992	126661	68331	53331
`mpdu`	`ftdc`	4 bis	144827	89996	54831	35998
`mpdu`	`ftdc`	4 ter	144994	66664	78330	36665

Table 4.8: Difference in cost for the "no free" tests

syntax, and twice slower than the light weight coding routine (Test 4 ter). As mentionned above, possible ameliorations include a better handling of memory allocation, and a better handling of error protections when decoding.

4.5.4 Conclusion

These series of tests showed that implementation techniques are as much important as design optimizations ("BER" decoding of MPDUs for Test 3 bis is faster than FTLWS decoding for Test 1). Several implementation oriented optimization still to be incorporated in the compiler (see section 4.5.5).

An other series of tests will be performed on the local network in order to measure the impact of the coding/decoding delay on the global performance of an application.

4.5.5 Deriving a more significative benchmark

The P1 protocol is rather complex and the MAVROS compiler generates about 12000 lines of C code, which takes several minutes to compile. We believed that there was room for more optimizations, but could not easily analyse the "execution profile" of such a big program; also, it took a rather long time to test a modification in the compiler. We needed to work on a simpler, yet realistic, test case.

In order to emulate the complexity of the P1 protocol, we choosed to use a "tree" structure:

- By varying the depth and the branching factor of the tree, we could adapt the benchmark to the profiles of various applications,
- By varying the type of information present in the leafs, we could mimic the mix of data types of these applications.

The analysis of our 19 X.400 messages yielded the following profile:

- The maximum "depth" of the ASN.1 encoding was 8,
- We had a total of 400 ASN.1 elements,
- 80 % of the elements were of a subtype of "OCTET STRING", 10 % STRING" and 10 % "INTEGER",
- The most frequent construct was "SET", then "CHOICE" and "SEQUENCE".

As a result, we modified our benchmark to include a new type of elements, called "Mix-Tree":

```
Mix-tree ::= SET {
                a CHOICE {
                        a INTEGER,
                        b BIT STRING,
                        c PrintableString},
                b OCTET STRING,
                c SEQUENCE OF Mix-tree OPTIONAL}
```

Then, we builded a test case which had the parameters listed above, and ran the tests. For conveniency reasons, we stopped using our old SUN-3s and performed the tests on a more powerful "DecStation 5000/200", and generated a first version of the coding and decoding routines, with the following performances:

	Coding time (μs)		Dec. time (μs)		Size (octets)	
	BER	FTLWS	BER	FTLWS	BER	FTLWS
Mix-tree	1953	1601	8788	4843	2918	6936

The coding and decoding routines generated by the MAVROS compiler for the "Mix-Tree" type were very short, and could easily be analysed "manually" or with the help of "profiling" tools. We tested a number of modifications in our ASN.1 BER coding and decoding algorithms:

- Speed up the encoding of tags and length field by using systematically the "indefinite encoding" form,
- Inline the encoding of "OCTET STRINGS",
- Revize the memory allocation for "SET OF" and "SEQUENCE OF" components in order to remove the need to compute "a priori" the number of the components,
- Use "header prediction" techniques to speed up the decoding of tags and that of length fields,
- Combine the look-up for "end of component mark" with the decoding of tags in "SET",
- Streamline the evaluation of "CHOICE" components when embedded in a "SET" or in another "CHOICE",
- Test of a new memory allocation scheme.

These optimizations resulted in an interesting performance gain: basically, the "BER" routine became faster than the non optimized "FTLWS" routines. After being tested, the optimizations were reported in the code generation programs of the "MAVROS" compiler, as well as in the "run-time" library, resulting in the performances listed in the following table:

	Coding time (μs)		Dec. time (μs)		Size (octets)	
	BER	FTLWS	BER	FTLWS	BER	FTLWS
Initial	1953	1601	8788	4843	2918	6936
Tags, etc	1429		3671		3268	
Memory	1429		2773		3268	
Compiled	1406	1367	3085	1679	3268	6936

If we analyse these results, we can observe that the compiled version:

- Provide better "coding" performances, as the optimisations were more radicals,
- Provide slightly slower "decoding" routines than the "hand-optimized" library, probably due to a more systematic insertion of syntax checks,
- Also improves the performances of the "FTLWS" decoding routines, mostly because of the better memory handling.

In order to verify the status of our optimizations, we ported the optimized compiler to our initial development machines, the SUN-3, and could compare the new results with the performances obtained for 19 MPDUS:

	Coding time (μs)		Dec. time (μs)		Size (octets)	
	BER	FTLWS	BER	FTLWS	BER	FTLWS
Initial	62330	37998	291821	243490	8647	18196
Revised	51664	35998	187325	144827		
Optimized	33498	36331	124995	115162	9731	18524

The important result here is that the BER performance is better than that of the light weight syntax for the coding procedure: the enhancement in the coding speed due to the optimizations in the MAVROS compiler allows to eliminate the gain of the light weight syntax, due to the smaller size of the BER encodings (9731/18524).

At this stage, we decided to pursue the testing further in order to compare the ASN.1 BER performances not only against our locally designed light-weight protocol, but also against a well tuned commercial standard, i.e. SUN XDR.

4.6 Performances measurements: BER vs XDR

4.6.1 XDR

XDR is a standard for the description and encoding of data. Except when absolutely necessary, the XDR data encoding does not use tag or length fields. Most data are sent in a standard format without any qualifying prefix. To avoid alignment problems during data conversions at both ends, the data types defined by XDR are a multiple of four bytes in length. The XDR refers to these four-octet units as blocks. The designers of the XDR data encoding tried to minimize conversion costs by assuming that the receiver knew what was being sent to it, which eliminates the need for most tags, and by using a combination of fixed data sizes and blocking factor (all data must be padded to a block boundary) to make the conversion more efficient in both directions. The XDR library is a collection of C-functions that convert data between local and XDR representations. Althougth it is quite possible to write the XDR-filters for a complex data types, it is easier to use RPCGEN, a compiler that accepts a specification written in the associated data description language called XDR Language , which is similar to the C-programming Language (XDR Data Description Language can be used only to describe data; it is not a programming language). From these specifications RPCGEN produces the XDR-filter routines.

4.6.2 The benchmark

The tests were performed using the same data types for BER and XDR. The ASN-1 specification of the "object" of the performance test is the following:

```
Perf-object ::= OPAQUE CHOICE {
mix[0] SEQUENCE OF Mix-tree}

Mix-tree ::= SET {
a CHOICE {
a INTEGER,
b PrintableString},
b OCTET STRING,
                c SEQUENCE OF Mix-tree OPTIONAL}
```

In order to perform the test with types whose complexity is similar to the complexity of MPDUS and that can be easily specified with XDR, we use recursive data structure.

The corresponding XDR specification of this object is the following:

Test identifier	Coding time	Decoding time	size
BER 50-50	19999	99496	7245
BER 100-0	16832	98496	4851
BER 0-100	21999	98496	9696
XDR 50-50	88000	225000	8640
XDR 100-0	75000	189000	6120
XDR 0-100	101000	265000	11220

Table 4.9: Results of the performance tests of both ASN.1 BER and XDR on a Sun 3/60 workstation

```
const MAX_STRING_LEN = 30; /* maximum length of a string */

enum utype {INTEGER = 1, STRING = 2};

union union_tag switch (enum utype utype) {
case INTEGER :
int a;
case STRING :
string b<MAXSTRINGLEN>;
};

/* A node in the tree */

struct Mix_node {
union_tag a;
string b<MAXSTRINGLEN>;
struct Mix_node son<>;
};
```

This data type is a tree structure with a depth of 8 and 255 nodes (just like the ASN.1 P1 MPDUs cited in the previous section).

4.6.3 The test functions

For the data type described above, MAVROS and RPCGEN will generate the C procedures for the coding and decoding functions respectively for the BER and the XDR syntax. For each syntax we performed the tests for the coding and decoding functions (cod and dec). These routines (cod, dec) will be repeated 100 iterations in a loop to obtain an average time per call just like the previous tests on BER ASN-1.

4.6.4 Performance Tests

We performed several series of tests on a SUN 3/60 workstation and on a DEC 5000/200 workstation. The results are shown in tables 4.9 and 4.10.

Where the $X-Y$ digits in the Test identifier field represent respectively the percentage of times the CHOICE field of the benchmark value is chosen to be an INTEGER or a PrintableString.

Test identifier	Coding time	Decoding time	size
BER 50-50	2499	4960	7245
BER 100-0	2070	4413	4851
BER 0-100	2851	4843	9696
XDR 50-50	7000	12000	8640
XDR 100-0	5000	10000	6120
XDR 0-100	8000	14000	11220

Table 4.10: Results of the performance tests of both ASN.1 BER and XDR on a Dec5000 workstation

4.6.5 Comments

These tests results allow us to make the following observations:

- The RISC processor on the DECstation reduced significantly the coding/decoding time. It is roughly the same enhancement for both BER and XDR encodings.
- BER showed better performance than XDR for all the test cases. Note that we used the optimized version of the Mavros compiler in these tests. This confrim our previous results concerning the comparison of BER and FTLWS: the implementation oriented optimization of the coding routines allow for a significant gain in the performance that should be considered before we decide to use alternative encodings.

The benchmark may not be suitable to show XDR optimized performance because it is somewhat complex.

4.7 Tests over the network

We performed series of tests over the network in order to measure the relative cost of the presentation layer for OSI applications. The tests consist in the exchange of data values of the tree structure used for the coding tests between two workstations (both SUN-3 and Dec5000) on the network using TCP-IP.

With this client/server model, two modes of operation were tested:

- a STREAM mode: consisting in the continuous transmission of data from the client, i.e. without waiting for any reply from the server.
- an "RPC" mode: where the client waits for the echo of his previous message before the transmission of the next one.

In the Stream mode the functions (coding, transmission, freeing) are repeated at the client for each message. At the server, the sequence (reception, decoding, freeing) is repeated until the end of the transmission. The corresponding sequences in the RPC mode are (coding, transmission, freeing, reception, decoding, freeing) and (reception, decoding, coding, reply, freeing) at the client and the server respectively. Each test consists in a loop of 100 iterations of these sequences and we measured the run time at the server. The reported time in the tables corresponds to the mean value for an iteration (time/i). This experimentation was conducted on both Ethernet and FDDI networks. The encoded messages were transmitted using the TCP-IP socket interface. In the next paragraphs we will present the results of the tests. We will also report the figures for "raw" TCP-IP (and UDP-IP) performance

Test identifier	time/i (ms)	size (octets)	throughput (kbps)
STREAM ASN.1	119	7245	480
STREAM XDR	282	8640	245
RPC ASN.1	253	7245	286
RPC XDR	451	8640	153
TCP-IP	8	3000	3000
UDP-IP	9.6	9000	7500

Table 4.11: Performance tests results of both ASN.1 BER and XDR between two SUN 3/60 workstations over Ethernet

Test identifier	time/i (ms)	size (octets)	throughput (kbps)
STREAM ASN.1	10	7245	5796
STREAM XDR	19	8640	3637
RPC ASN.1	35	7245	1656
RPC XDR	51	8640	1355
TCP-IP	2.74	3000	8760
UDP-IP	1.22	1400	9180

Table 4.12: Performance tests results of both ASN.1 BER and XDR between two Dec 5000 workstations over Ethernet

i.e. without any encoding at the presentation level in order to evaluate the cost of the presentation layer itself. The TCP-IP figures were obtained with classical client/server socket "stream" interface: the client establishes the TCP connection and then transmits the data it has to send. The server will enter a loop until all the messages have been received. Only the Dec stations are connected to the FDDI network.

4.8 Results and comments

The figures are reported in tables 4.11, 4.12, 4.13. We have the following comments:

- The results for SUN3 over Ethernet showed bad performance for both ASN.1 and XDR: this limitation is due to the processor speed which can not perform all the coding/decoding routines rapidly. This is confirmed by the TCP-IP figure in table 4.11:

Test identifier	time/i	size (octets)	throughput (kbps)
STREAM ASN.1	8	7245	7245
STREAM XDR	17	8640	4065
RPC ASN.1	28	7245	2070
RPC XDR	44	8640	1570
TCP-IP	1.44	3000	16666
UDP-IP	1.21	3000	19834

Table 4.13: Performance tests results of both ASN.1 BER and XDR between two Dec 5000 workstations over FDDI

the SUN3 machine can transmit at a maximum throughput of 3 Mbps, the cost of the presentation layer is very high relatively to the transport.

- The Dec5000 stations showed completely different results over Ethernet. The ASN.1 and TCP-IP figures are comparable: ASN.1 is only 1.5 times slower than the "raw" transport. The increase in the processor speed allowed for a better performance for the stream mode: to some extent, ASN.1 encoding could be streamlined with the transport without significant loss in the performance. These results apply to FDDI with a lesser extent. In fact, the figures with ASN.1 showed only 43% (7.2Mbps/16.6Mbps) of raw TCP performance. The relative cost of the presentation layer increases with the network throughput. However, previous experiments (not reported in this document) showed worst performance for ASN.1, the optimizations of the Mavros compiler led to reduce the cost of the presentation.

- ASN.1 always showed better performance than XDR. This is due to the successive optimizations of the compiler. Even on the SUN 3 machines the XDR figures does not allow to saturate the available bandwidth of 3 Mbps (at the TCP level).

- The maximum acheivable throughput on the FDDI network interface is about 17 Mbps at the TCP level. This result outline the need for high speed transport protocols like TPX in order to allow the available bandwidth to be propagated to the application level.

- The RPC performance are independant from the coding syntax (ASN.1 or XDR) and the network speed (Ethernet or FDDI). The mdel itself is not very suitable for operation at high speeds.

4.9 Conclusion

The light weight syntax always shows better performance than the Basic Encoding Rules. However, the performance gain we have by using such syntax is minimized if the coding routines are optimized. As these software optimizations enhance considerably the performance for BER the difference in the coding/decoding speed between BER and FTLWS is reduced. For many times, the "optimized" version of BER had better performance than "old" version of the FTLWS coding functions. On the other hand optimized versions of the BER routines have better performance than XDR. The results in the next chapter confirm this idea: one should better concentrate on the optimization of the standard encoding rules rather than to propose alternative encodings or hardware implementation.

The network tests confirmed that the cost of the presentation layer is preponderant when the ration processor/network speed is reduced. However, successive optimizations of the mavros compiler reduced considerably this cost for ASN.1 BER and one could expect to reach the raw transport limitation with software optimizations and increased processor speeds.

Chapter 5

Testing an optimized BER library

5.1 INTRODUCTION

5.1.1 Purpose

This chapter contains the definition of some ASN.1 benchmarks and their application to some current and hypothetical encoders/decoders. These include an optimized version of the DIT implementation of BER (*liba1* [33, 34]), the widely used ISODE cookbook (*posy* [Ros87, Ros88]), and a decoder based in an hypotetical BER chip outlined in [32].

These benchmarks try to determine how fast BER are and how much tool independent is their speed. In the following section the benchmarks are designed, then the performance measurements are presented and commented and, finally some conclusions are drawn.

5.1.2 Definitions

ASN.1: ISO Abstract Syntax Notation One.
BER: Basic Encoding Rules for ASN.1.
RISC: Reduced instruction set computer.
CISC: Complex instruction set computer.
i/i: Number of instructions/integer.
HS: High speed networks.

5.2 DESIGN

5.2.1 What is to be measured

For HS the most important feature of a transfer syntax is how long does it take to fully encode and decode data values. However, in some cases (such as e-mail) the application code is only interested in a small part of the data, realying or throwing away the rest. The cost of this kind of operation should be taken into account too.

Before proceeding it is mandatory to define what do we mean as decoding and encoding. The common assumption is a mapping between internal data structures and external bit streams. However, this mapping may lead to the building of a complex internal data structure just to be able to process a relatively simple external data stream. The cost of building and freeng the structure may be a lot higher than decoding the data items

themselves. So another point of view of decoding is the work needed to get individual data items in a register.

Another factor on decoding/encoding speed is the amount of checking performed on redundant data. The basic encoding rules carry a lot of redundancy. The more checking, the less speed, of course.

Once decided what is encoding/decoding, the influence of different aspects of data on speed must be clearly stated. For example, the influence of data complexity and shape may be important, even for the same number and values of elementary items. For BER, important aspects are the influence of the number of tags, the length form (definite or indefinite), and whether strings are primitive or constructed.

5.2.2 Simple types

Most benchmarks are made out of the ASN.1 INTEGER type, because it is one of the more used and it is complex. Of course, REALs are more complex, but rarely used. For INTEGERs, measurements are made for small (1) and big ($2^{31} - 1$)) values. The type chosen to measure INTEGER costs is a simple SEQUENCE:

```
SeqTbl ::= SEQUENCE {
             int1 INTEGER,
             int2 INTEGER,
                  .
                  .
             intn INTEGER
           }
```

This type is very simple. Almost all of processing time is due to the the component processing, so we can estimate the processing cost of INTEGER dividing the whole by the number of components.

5.2.3 Complex types

The processing cost of complex types is larger than for simple ones because the coding overhead added is greater. To measure the cost of complexity we can encode and decode the same integer values of the previous section, but including them in a more complex structure:

```
NestTbl ::= SEQ {
             int  [1] IMPLICIT INTEGER,
             nest [2] IMPLICIT NestTbl OPTIONAL
            }
```

Notice that each integer in a *SeqTbl* has one ASN.1 identifier associated, while in a *NestTbl* each integer has two.

When some ASN.1 types (CHOICE, SET, SEQUENCE with OPTIONALs) are decoded, different components may arrive at a certain moment, therefore the decoder has to make checkings to determinate what component is arriving. These checkings increase the decoding cost of complex types. To measure this increment we use the next type, that is similar to SeqTbl, but it allows any arrival order:

```
SetTbl ::= SET {
             int1 [1] IMPLICIT INTEGER,
             int2 [2] IMPLICIT INTEGER,
```

```
    .
    .
  intn [n] IMPLICIT INTEGER,
}
```

5.2.4 The influence of the length form

The BER allow to encode the length of constructed value in two forms. The first form is to encode the byte number of the value (definite form) and the second is to mark the end of content of the value (indefinite form). The definite form is usually more memory efficient than indefinite form, but its calculation is harder and the whole data value has to be in memory while encoding. The type *NestTbl* is suitable to measure the influence of length form on encoding/decoding cost.

Sometimes the decoder does not want to decode completely the data, but skip useless parts or to relay them. If definite form is used skipping and relaying can be done very efficiently, without examining all bytes. Indefinite form, however, forces to examine all them. The type *NestTbl* is again suitable to measure the influence of length form on skip and copy operations. We can get a value of this type, to decode the first component and then skip or copy the remainder.

5.2.5 The cost of type checking

The BER external format carries out some sort of type information in tags. Sometimes the decoder needs to look at these tags in order to decode the arrived component, but other times it knows exactly what component has to arrive and to check the tag is redundant. Checking tags may be interesting to allow type extensions, to fix bugs in development phase, or to be more robust against misbehaved partners. The cost of checking tags is measured removing redundant type checkings in types *SeqTbl* and *SetTbl*.

While checking tags in not a very costly operation, checking other components of the type, such as complex constranits, may be more costly. The cost of this kind of checkings has not been measured.

5.3 TARGETS

The tools used as benchmark targets were the following:

- An optimized version of the DIT implementation of BER (*liba1*), which is a library designed to easily cope with BER like encodings, hiding the user the complexities of lengths and primitive/constructed bit handling.
- An hypotetical implementation of the above using a presentation chip outlined in [32]. The chip is supposed to implement almost *liba1* interface with a slower technology than the main processor (this is the most common situation for specialized coprocessors).
- The widely used ISODE cookbook, as a reference point. It is consist of a set of translators (*posy*, *pepy*), from (annotated) ASN.1 to the cookbook library calls. Its workings are radically different from those of *liba1* and used for comparison.

5.4 ENVIRONMENT

ISODE benchmarks have been made on a Sun 4/370 (CISC, 4 MIPS) and *liba1* benchmarks on a Sun 4/370 and on a Sparcstation1 (RISC, 12.5 MIPS). The operating system used

has been SunOS 4.1.1 The compilations were made with Sun's *cc* compiler with the *-O* optimization option.

To obtain the processing cost the *getrusage(2)* function has been used. This function has a 1/50 second resolution on Sun-3 and a 1/100 second resolution on Sun-4. Each measurement has been performed 10000 times, and the mean value is reported. The benchmarks were performed while other users were working on the machines. The estimated error is about 10 per cent (note: The error was estimated repeating each benchmark several times and computing the difference between the main value and the furthest value from it).

In order to be as tecnology independent as possible, measurements are reported in *normalized instructions*, that is, the time in microseconds multiplied by the official MIPS figure.

For *liba1* and *ISODE*, the benchmarks have measured three kinds of costs: encoding, decoding and freeing costs. To interpret these costs correctly, the features of each tool have to be taken into account:

- *posy* generates encoding, decoding and freeing routines (if *-f* option is selected) from an ASN.1 type definition. The encoding cost is the necessary one to generate the BER format of a user supplied value. The cost necessary to create and fill this value is not included. The *posy* decoding routines always allocate the memory necessary to store the decoded value. The decoding cost includes the cost associated with this allocation and with translation from BER format to internal format. The cost necessary to free allocated memory for decoded value is the freeing cost. This time is only significant for unlimited size types (SETOF, SEQUENCEOF and recursive types).

  The ISODE routines have been generated with the following options: *-h* (heuristic) and *-f* (to generate freeing routines) *posy* options and *-r* (to check type information for extensibility) *pepy* option.

- *liba1* have designed to process and produce data *serially* so it has been used in such way, using hand-coded routines. These routines process data on fly, so decoding routines do not allocate memory and there are not freeing routines. Therefore decoding cost does not include allocation and freeing cost is always zero. For the tests *liba1* has been generated with the following *high speed* generation options:

  - *FASTCARRIER*, to encode/decode only in memory.
  - *FIXSTACK*, to use a fix size stack for open constructors.
  - *NDEBUG*, for not checking assertions.
  - *SMALLTAGS*, identifier numbers smaller than 30.

Finally, the hypotetical chip tests have been made by C compiler generated code examination, replacing *liba1* calls by direct chip instructions. The chip was supposed to be slower than the CPU, lasting 15 main CPU instructions to decode/encode a header (tag and length) and 10 instructions to encode/decode a 32 bit integer. However the chip works in parallel with the main CPU and predecodes headers and contents usually well before the main CPU issues the instructions. These calculations were made only to *SeqTbl*.

5.5 RESULTS

5.5.1 Simple types

Only *SeqTbl* has been tested for the chip and the two extreme integers. For comparison purposes, the results of the previous non optimized version of *liba1* are included too.

SeqTbl	values	encoding (i/i)	decoding(i/i)
liba1 (RISC)	1	224	258
	2147483647	286	301
liba1 (CISC)	1	306	259
	2147483647	328	296
non optimized liba1 (RISC)	1	1290	1310
	2147483647	1352	1355
non optimized liba1 (CISC)	1	1005	999
	2147483647	1030	1041
liba1 with chip (RISC)	1	40	47
	2147483647	40	47
posy (CISC)	1	1248	1125
	2147483647	1164	1284

concluding:

- The optimization of *liba1* has been very important.
- The increment of cost between 1 and 2147483647 values is only 28 % for optimized *liba1* routines, and zero for *posy* routines and obviously the chip.
- Costs are only slightly bigger for RISCs.
- The *posy* routines are four times slower than *liba1* ones. On the other side, using the chip speeds up processing by almost one order of magnitude.

Other benchmarks have been made using context [256] tags instead of INTEGER's tags for components. The encoding cost was incremented 75 % and decoding cost 16 %. These increments are mainly due to the SMALLTAGS option had to be disabled. However big identifiers greater than one byte are unusual in common ASN.1 types.

5.5.2 Complex types

The *NestTbl* type has been encoded/decoded only for 1 in the integer components.

NestTbl	encoding (i/i)	decoding(i/i)	freeing(i/i)
liba1 (RISC)	645	694	0
liba1 (CISC)	557	518	0
posy (CISC)	1897	2219	328

concluding:

- This cost is almost three times greater than *SeqTbl* cost on RISC and twice on CISC (for both *liba1* and *posy*).
- The *posy* freeing cost is about 10 % of decoding cost.
- The ratio between *liba1* cost and *posy* cost is similar for *NestTbl* and *SeqTbl*. However if the freeing cost is included in the decoding cost *liba1* decoding is five times faster than *posy*'s.

Also the *SetTbl* has been benchmarked. The value of each component has been 1 too.

SetTbl	encoding (i/i)	decoding(i/i)
liba1 (RISC)	260	331
liba1 (CISC)	321	363
posy (CISC)	1292	1215

concluding:

- The encoding cost is similar to *SeqTbl* encoding cost, because no additional checkings are made encoding sets.
- For *liba1*, *SetTbl* decoding time is 28 % greater than *SeqTbl* decoding cost on the RISC and 48 % greater on the CISC machine.
- The *posy SetTbl* decoding cost is similar to *SeqTbl* one.
- The *liba1* decoding routine is three times faster than *posy* routine.

5.5.3 Type checking

To measure the type checking cost all redundant checkings have been removed from *SeqTbl* and *SetTbl liba1* decoding routines. So the *SeqTbl* decoding routine does not check any identifier and *SetTbl* decoding routine does check if a component arrives twice or a mandatory component is missing. Some internal consistency checkings have not been removed. The decoding costs with and without type checking happened to be almost identical.

For *posy* routines this cost is also negligible. The *posy* routines were generated without *-r* (robust code) option and the cost was slighty incremented!

5.5.4 Length form

The *NestTbl* type has been encoded/decoded using different length strategies, for constructed data. *Posy* has three strategies: default (the length that fit into one byte is encoded as definite, otherwise as indefinite), definite (lengths are encoded as definite) and indefinite (lengths are encoded as indefinite).

The *liba1* library always encodes lengths in indefinite format. Therefore only decoding cost can be measured for the definite length form.

NestTbl	encoding (i/i)	decoding(i/i)	freing(i/i)
liba1 definite (RISC)	—	791	0
liba1 indefinite (RISC)	645	694	0
liba1 definite (CISC)	—	481	0
liba1 indefinite (CISC)	557	518	0
posy definite (CISC)	1704	2100	279
posy default (CISC)	1897	2219	328
posy indefinite (CISC)	1947	2302	229

There are not big differences between strategies. However is curious to note that for *posy* the definite strategy is the fastest one.

The length encoding strategy has greater influence when data is not completely decoded, but skipped or copied. The next results show the cost of copying and skipping a *NestTbl* compared to decoding it (in instructinons per integer copied, skipped or decoded). Only *liba1* library could be used to do this benchmarks, skipping or copying 100 integers.

NestTbl	i/i
skip definite (RISC)	25
skip indefinite (RISC)	709
copy definite (RISC)	38
copy indefinite (RISC)	1338
decode definite (RISC)	791
decode indefinite (RISC)	694

concluding

- Skipping and copying definite length data is much faster than for indefinite length data.
- For small values and indefinite length, skipping cost is similar to decoding cost and coping cost to decoding plus encoding cost.

5.6 CONCLUSIONS

The more important conclusions of these measures is that BER handling is costly. Existing tools can be greatly enhanced as demonstrated by a relatively strightforward optimization of our library. Further improvements can be made with architectural changes, as demosntrates INRIA's tool [7]. However the limit may be the limit placed by a fast hardware chip. Even in this case the results are not better than those using a simple, fixed length encoding scheme, without hardware support.

Other important and expected conclusions are that tagging has an important cost, that the buiding of internal structures may account an important part of decoding time (sometimes much more important), and that he use of definite lengths improves the blind skipping and copying of components, at the cost of a high encoding complexity. However, once tagging is used, the cost of checking them is negligible.

Chapter 6

The presentation protocol

6.1 Introduction

6.1.1 Purpose

Assessment on ISO presentation service and protocol to check their suitability for High Speed and Multi Media applications. Coding rules aspects of presentation are not covered because they are assesed in [32].

6.1.2 Definitions

ASN.1: ISO Abstract Syntax Notation One.
HS: High Speed applications.
MM: Multimedia applications.
BER: Basic Encoding Rules for ASN.1.
PER: Packed Encoding Rules for ASN.1.
DER: Distinguished Encoding Rules for ASN.1.
CER: Confidential Encoding Rules for ASN.1.
PPDU: Presentation Protocol data unit.

6.2 Concepts

Distributed applications often cooperate with components hosted on different computer architectures. These computer architectures use different ways of representing data items (numbers, strings, images, sound, video, etc) and structures (records, sets, etc), called *internal data formats*. So they have either to know the way their partners represents them, or they have to *negotiate* a common representation and convert all data to this representation for the transfers.

Common representations of data items are called *external data formats* or *transfer syntaxes* and common descriptions of data types are called *abstract syntaxes*.

Because applications are sometimes made out of several application service elements with their own types, and because these service elements could be made by different software manufacturers, negotiations should be made in terms of sets of abstract syntaxes and the corresponding transfer syntaxes. These pairs are called *presentation contexts*, and the whole set is called a *context set*.

6.3 Impact of the presentation negotiation to HS and MM

From [32] we know the heavy impact of encoding and decoding on distributed applications performance. Negotiation of rules may improve the performance only if it leads to a representation akin to one or both peers. However the very fact of negotiability has a cost implying a negotiation process and the implementation of multiple rules. On the other side connectionless applications can not negotiate; other applications, like electronic mail, cannot rely on the presentation protocol negotiation.

From [32] we saw that the reward of negotiation is not impressive for rules well designed for speed and machines able to communicate with every other machine in the world. However, for machines restricted to speak with similar ones or with *multilingual* machines, the gain is very important.

Multimedia applications are more demanding for types like voice, image or moving video, so the negotiation of their encodings may be crucial for their interoperability. The interoperability of heterogeneous machines may well require the cooperation of a third machine having a good transcoder.

6.4 Overview of ISO presentation service and protocol

There is one ISO standarized connection oriented presentation service [21] and protocol [20], though a connectionless one [29, 30] is being standarized.

6.4.1 Connection oriented presentation

For two applications to work together they have to agree first about the data types that they need to exchange. This agreement is reached by means of a negotiation between the applications about the abstract syntaxes that can be handled by both applications and by the presentation provider. The presentation service provides primitives to make this negotiation and to change the set of agreed abstract syntaxes during the connection. Besides these services, the presentation layer provides all the session services.

This negotiation is performed in the following way. When an application opens a connection, it passes a list of abstract syntaxes to the local presentation entity (the presentation context definition list). These abstract syntaxes are identified by registered unique *object identifiers*. Each abstract syntax is associated with a local identifier called *context identifier*. This identifier will be used by the application to indicate to the presentation entity the abstract syntax to which each user data to be sent belongs.

Following the connection request, the local presentation entity makes a new list and sends it to the remote presentation entity. Each item of this list contains a context identifier, an abstract syntax and a list of the transfer syntaxes that support the abstract syntax and are provided by the local presentation entity. From this list the remote presentation entity generates an abstract syntax list marked as accepted or rejected. The remote entity will mark as rejected those abstract syntaxes that it can not support using one of the proposed transfer syntaxes. Then they are passed to the remote application that can mark other abstract syntaxes as rejected. For accepted ones the remote presentation entity has to select one of the proposed transfer syntaxes. The result of the selection is returned to the initiating presentation entity that informs its user which abstract syntaxes are accepted and which are rejected.

If the negotiation has been successful both applications know a list of accepted abstract syntaxes (the presentation context definition result list). Besides, both presentation entities know which transfer syntaxes will be used for each accepted abstract syntax. Each pair of

abstract syntax and transfer syntax is called a *presentation context* and the list the *defined context set*. Abstract syntaxes can be added and removed from the defined context set during presentation connection if the *context management functional unit* has been selected. The presentation protocol guarantees that the defined context sets of both entities are consistent.

If the presentation user selects the *context restoration functional unit*, the user is allowed to get back to previous connection states marked by *synchronization points*. To do this, when a presentation entity receives a request or an indication for a synchronization point, it stores the current defined context set together with the point number and requests the session provider corresponding service. Then when a presentation entity receives a request or an indication to resynchronize to one point, it has to restore the associated defined context set and to pass the request to the session provider or to send an indication to the presentation user.

6.4.2 Connectionless presentation

Proposed connectionless presentation is much simpler than connection oriented one, because there is not negotiation. Each presentation data unit carries a context set for the application data, so the receiver has all information needed to perform the decoding, if possible. In order to increase the chance of the receiver to be able to decode the data, the sender may encode it in several transfer syntaxes.

6.5 Presentation layer problems related to speed

The main problem of the presentation layer regarding speed is encoding and decoding, as stated in [32], so the development of fast rules and close up negotiations is vital. An associated role assigned to presentation is to do type checking, so no conformant data will cause an abortion of the connection. Type checking is costly, specially if constraints are considered part of the type.

A related problem is that the control part of PPDUs are encoded using ASN.1 BER. We know that BER are computationally costly, so they may be a problem. In fact, the so called *fully encoded data* mode for *user data* is frequent and relatively complex, so a bunch of instructions may be lost on them. There are other PPDUs much more complex, but hopefully less frequent.

The last problem is the very fact that presentation is another layer (perhaps placed in the wrong place), with its state machine and the need to pass data between levels. Only good implementation techniques may reduce layer overhead, perhaps collapsing layers in implementations. In fact, functions supposed to be at this layer cannot be placed in a straigthforward way. Presentation and application layer software implementations can not be as independent as Reference Model suggests, because it will be inefficient and difficult to maintain. Let us see why and outline solutions.

Because applications conceptually pass abstract syntax names to presentation, this layer must maintain a database of syntaxes, giving its description and applicable transfer syntaxes. So, the presentation level should be changed for each new application supported. The real solution is to somehow pass this information from the application.

If presentation gets type information, either from the database or from the application, to be able to perform the decoding, it will be time inefficient, because of the interpretation of the type. On the other side, presentation does not know how the application wants the decoded data, so it should pass a description of the internal data type(s) to presentation too, leading to an even more inefficient system. The solution is to let the application perform the decoding (and encoding), so the only thing presentation does is to offer an encoding

and decoding library for the rules it supports. Even an off line tool such an *ASN.1 compiler* may be regarded as part of the presentation level.

Using this approach, aplications offer the encoding rules, not presentation. Consequently, in order to be able to *speak* several rules, applications must have encoding and decoding routines for all them, wasting memory and (perhaps) programming effort.

An approach to save memory and let presentation choose the encoding rules is to define a programming interface able to encode and decode any datum described by a type in a given data definition language. This interface may be dynamically bound to the actual encoding rules with a small loss of efficiency. It is also suitable for *fine grain negotiation* as we will see next.

6.6 Fine grain negotiation

Current presentation negotiation procedure is too simple to cope with the task of adapting similar architectures in order to improve efficiency. In fact it is too simple to handle tasks such as compression and encryption, traditionally placed at presentation level (this is currently being discussed, because the effectiveness of compression depends upon the nature of data, and cryptography is not, strictly speaking, a presentation task). The main problem is that presentation negotiates only abstract syntax and transfer syntax *names*. And names cannot express a great variability: *adjectives* and perhaps *complex descriptions* are more expressive.

Let us suppose we want to send a structured ASN.1 value compressed and then encrypted. Its transfer syntax may be the addition of say BER plus a Lempel-Ziv compression algorithm plus DES with an agreed key. An alternative may be the use of DER plus a Huffmann coding with a precalculated frecuency table plus DES again. These alternatives cannot be expressed with a single registered name.

Names are not always appropriate for abstract syntaxes either, because sometimes applications need to work with dynamic types. However this is neither frequent nor appropriate for high speed networks.

More relevant to high speed and multimedia is the separate negotiation of encodings for simple data and structuring methods. A form of separate negotiation is already in use for different abstract syntaxes, so an extension of the method may be devised. However, is simpler to propose transfer syntax families.

For example, a powerful machine may propose a fixed untagged binary encoding (see [32]) with either 16 or 32 bits integers big or little endian, with IA5 or EBCDIC text strings, etc., optionally compressed with a Lempel-Ziv or a Huffmann algorithm. The other may accept 16 bits little endian, IA5, etc., uncompressed. Of course, a variable tagged encoding may be offered too, in another transfer syntax family. The order of the alternatives may express the order of preference of the initiator.

Fine grain negotiation may be particularly useful for multimedia data, where the cost of decoding may be very high. However, these types may be considered EXTERNAL, and thus belonging to another context, rendering fine grain negotiation less interesting.

Of course, parameters agreed along with the transfer syntax name will form part of the presentation context. Moreover, if some transfer syntax would have memory across different data units (adaptive compression may be a good example), the status should belong to the context. If the context restoration functional unit is selected, these parameters and statuses must be saved and restored with the rest of context.

6.7 Conclusions

ISO presentation does not meet the goal of choosing the best encodings for similar machines. BER are costly because instead of being an Esperanto of the different machines (taking the most widespread encodings methods), define a new language strange for all them. Even the presentation level is difficult to implement in the way it is conceptually defined, being many of its tasks actually carried by applications directly.

A fine grain negotiation mechanism has been outlined to close the distance between different machines. It can be implemented without excessive cost, specially for multimedia data. However, standardization is always better than negotiation, so the less things are negotiated, the better.

Chapter 7

Conclusion

Within this task we performed detailed assessment of both session and presentation layers focusing on the cost of the encoding/decoding routines at the presentation level. We can make the following remarks:

- The synchronization functions of session layer can reduce considerably the throughput achieved by the upper layers. This should be resolved by a proper design of the synchronization functions.
- ASN.1 is suitable as a data description language. The inclusion of the Macros facilty within the Mavros compiler allow to extend its capabilites. An interesting subject is to interface ASN.1 with LOTOS. This subject is left for further research.
- The results of the light weight encodings are not very interesting: implementation oriented optimizations showed similar performance for both BER and FTLWS.
- A fine grain negotiation mechanism may be implemented without excessive cost and will allow to reduce the distance between different machines.
- Implementation optimizations led to ASN.1 performance better than XDR.
- Hardware support for the presentation layer does not show big benefits that cannot be acheived with software optimizations.

The conclusion of this work is that BER encoding should be supported as it may be implemented with reasonnable cost over high speed networks. The need for a global architecture for the OSI upper is a more general problem that should be addressed elsewhere. However, the optimized version of the presentation should lead to the implmentation of the presentation filter in order to acheive "streamlined" encoding and transmission operation of OSI applications.

Contents

Bibliography

[1] Adobe Systems Iincorporated. *PostScript Language Reference Manual.* Addison Welsley, Reading, Mass., 1987.

[2] John K. Bennet. Experience with distributed smalltalk. *Software–Practice and Experience*, 20(2):157–180, February 1990.

[3] A. Black, N. Hutchinson, and H. Levy. Object structure in the emerald system. In *OOPSLA 86*, pages 78–86. ACM, Sept 1986.

[4] David H. Crocker. Standard for the format of ARPA internet messages. *Internet Working Group Requests for Comments*, (RFC 822), August 1982.

[5] L. Heuser et al. Language constructs to express distribution of object-oriented applications. In *TOOLS 89*, pages 355–362. Angkor, November 1989.

[6] Ulf Hollberg et al. An object oriented view of distribution. Technical Report 43.9004, IBM-ENC, 1990.

[7] Christian Huitema and Assem Doghri. Defining faster transfer syntaxes for the osi presentation protocol. *Computer Communication Review*, 19(5):44–55, October 1989.

[8] ISO. Specification of abstract syntax notation one (ASN.1). *Information Processing Systems: Open Systems Interconnection*, (IS 8824), 1987.

[9] ISO. Specification of abstract syntax notation one (ASN.1), addendum 1: ASN.1 extensions. *Information Processing Systems: Open Systems Interconnection*, (DIS 8824 - DAD 1), 1988.

[10] ISO. LOTOS - a formal description technique based on the temporal ordering of observational behaviour. (IS 8807), 1989.

[11] Cosmos Nikolau. An architecture for real-time multimedia communication systems. *IEEE Journal on Selected Areas in Communications*, 8(3):391–400, April 1990.

[12] J. Postel. Internet multimedia document format. *USC Information Sciences Institute*, (MMM-12 (RFC 767-Revised)), March 1982.

[13] Sun Microsystems, Inc. *Networking on the Sun Workstation. External Data Representation: Protocol Specification.* 2550 Garcia Avenue, Mountain View, CA 94042, USA, 1986.

[14] Niklaus Wirth. From Modula to Oberon. *Software-Practice and Experience*, 18, 1988.

[15] Xerox Corporation. Courier: The remote procedure call protocol. *Xerox OPD, 3333 Coyote Hill Rd, Palo Alto, Ca 94304*, (XSIS 038112), December 1981.

[16] Xerox Corporation. Courrier. Xerox System Integration Bulletin, OPD B018112, 1981.

[17] Sun Microsystems, Inc., XDR: External Data Representation, 1987.

[18] T.H. Dineen and al. "The Network Computer Architecture and System: An Environment for Developing Distributed Applications". Summer USENIX Conference. Phoenix, Arizona, 1987.

[19] C. Partridge, M.T. Rose. "A Comparison of External Data Formats". IFIP International Working Conference on Message Handling Systems and Distributed Applications. Costa-Mesa, California, 1988.

[20] ISO. Connection oriented presentation protocol specification. *Information Processing Systems: Open Systems Interconnection*, (IS 8823), 1988.

[21] ISO. Connection oriented presentation service definition. *Information Processing Systems: Open Systems Interconnection*, (IS 8822), 1988.

[22] ISO 8824 Specification of Abstract Syntax Notation One (ASN.1).

[23] ISO 8825 Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).

[24] CCITT X.409 - Red book, fascicle VIII.7, Torremolinos, 1984.

[25] Michel Besson, Assem Doghri. "High Performance Heterogeneous Transmission using the OSI Presentation Protocol". ISCIS 3. Izmir, Turkey, 1988.

[26] Christian Huitema, Assem Doghri. "A High Speed Approach for the OSI Presentation Protocol". IFIP Workshop "Protocols for High-Speed Networks". Zurich, Switzerland, 1989.

[27] IEEE Standard for binary floating-point arithmetic.

[28] Christian Huitema. Definition of the Flat Tree Light Weight Syntax (FTLWS). Internal Document, INRIA Sophia Antipolis, July 1990.

[29] ISO. Connectionless mode presentation service definition. *Information Processing Systems: Open Systems Interconnection*, (8822/DAD1), 1990.

[30] ISO. Connectionless oriented presentation protocol to provide the connectionless mode presentation service. *Information Processing Systems: Open Systems Interconnection*, (DIS 9576), 1990.

[31] OSI-95. High performance OSI protocols with multimedia support on HSLAN's and B-ISDM. Detailed project description. Technical Report OSI 95–TA–II.2, ESPRIT, 1990.

[32] J. Seoane and M. A. Lombardero. Report on data transparency and transfer syntaxes for high speed. Technical Report OSI95/DIT/C1/2/TR/R/Version 2, DIT, May 1991.

[Ros87] Marshall T. Rose. ISODE: Horizontal integration in networking. *ConneXions*, May 1987.

[Ros88] Marshall T. Rose. *The ISO Developement Environment User's Manual.* The Wollongong Group, 1129 San Antonio Road, Palo Alto, CA 94303, USA, 1988.

[33] Joaquín Seoane and M. A. Lombardero. The medium level ASN.1 library tutorial. *Internal Working Document*, 1990.

[34] J. Seoane and M. A. Lombardero. The liba1 library tutorial. Technical Report OSI95, DIT/C1/5/TR, Version 0. OSI95 ESPRIT II Project, October 1991.

Appendix A

The MPDU type definition (From X.411, 1984).

```
MPDU ::= CHOICE{[0] IMPLICIT MPDU-Utilisateur, MPDU-Service}

MPDU-Service ::= CHOICE{[1] IMPLICIT MPDU-Rapport-Remise,
[2] IMPLICIT MPDU-Essai}

MPDU-Utilisateur ::= SEQUENCE{Enveloppe-UMPDU, Contenu-UMPDU}

Enveloppe-UMPDU ::= SET {
       Identificateur-MPDU,
       expediteur Nom-OR,
       initiaux Types-Codage OPTIONAL,
       Type-Contenu,
       ID-UA-Contenu OPTIONAL,
       Priorite DEFAULT normal-priority,
       Indicateur-Message DEFAULT {},
       remise-Differee [0] IMPLICIT Date-Heure OPTIONAL,
               bilateral[1] IMPLICIT SEQUENCE OF
Info-Bilaterales-Domaine OPTIONAL,
       destinataires[2] IMPLICIT SEQUENCE OF Info-Destinataire,
       Informations-Trace}

Contenu-UMPDU ::= OCTET STRING

Date-Heure ::= UTCTime

Identificateur-MPDU ::= [APPLICATION 4] IMPLICIT SEQUENCE {
   Identificateur-Global-Domaine, IA5String}

Type-Contenu ::= [APPLICATION 6] IMPLICIT INTEGER {p2(2)}

ID-UA-Contenu ::= [APPLICATION 10] IMPLICIT PrintableString

Priorite ::= [APPLICATION 7] IMPLICIT INTEGER {
  normal(0), non-Urgent(1), urgent(2)}

normal-priority Priorite ::= normal
```

```
Indicateur-Message ::= [APPLICATION 8] IMPLICIT BIT STRING {
       divulgation-Destinataires(0),
       conversion-prohibee(1),
       destinataire-Suppleant-Autorise(2),
       demande-Renvoi-Contenu(3)}

Info-Bilaterales-Domaine ::= SEQUENCE {
     Nom-Pays,
     Nom-Domaine-Administratif,
     Info-Bilaterales}

Info-Bilaterales ::= ANY

Info-Destinataire ::= SET{
      destinataire Nom-OR,
      [0] IMPLICIT Identificateur-Complementaire,
      [1] IMPLICIT Indicateur-Destinataire,
      [2] IMPLICIT Conversion-Explicite OPTIONAL}

Identificateur-Complementaire ::= INTEGER

Indicateur-Destinataire ::= BIT STRING

Conversion-Explicite ::= INTEGER {texte-iA5-Teletex(0), teletex-Telex(1)}

Informations-Trace ::= [APPLICATION 9] IMPLICIT SEQUENCE OF
       SEQUENCE {Identificateur-Global-Domaine,
       Info-Fournies-Domaine}

Info-Fournies-Domaine ::= SET{
      arrivee [0] IMPLICIT Date-Heure,
      differee [1] IMPLICIT Date-Heure OPTIONAL,
      action[2] IMPLICIT INTEGER{relaye(0), reroute(1)},
      converti Types-Codage OPTIONAL,
      precedent Identificateur-Global-Domaine OPTIONAL}

Identificateur-Global-Domaine ::= [APPLICATION 3] IMPLICIT SEQUENCE {
  Nom-Pays,
  Nom-Domaine-Administratif,
  Identificateur-Domaine-Prive OPTIONAL}

Nom-Pays ::= [APPLICATION 1] CHOICE{
     NumericString,
     PrintableString}

Nom-Domaine-Administratif ::= [APPLICATION 2] CHOICE{
             NumericString,
             PrintableString}

Identificateur-Domaine-Prive ::= CHOICE {
         NumericString,
 PrintableString}

Nom-OR ::= [APPLICATION 0] IMPLICIT SEQUENCE{
Liste-Attributs-Standard,
```

```
Liste-Attributs-Definis-Domaine OPTIONAL}

Liste-Attributs-Standard ::= SEQUENCE {
     Nom-Pays OPTIONAL,
     Nom-Domaine-Administratif OPTIONAL,
     [0] IMPLICIT Adresse-X121 OPTIONAL,
     [1] IMPLICIT ID-Terminal OPTIONAL,
     [2] Nom-Domaine-Prive OPTIONAL,
     [3] IMPLICIT Nom-Organisation OPTIONAL,
     [4] IMPLICIT Identificateur-Unique-UA OPTIONAL,
     [5] IMPLICIT Nom-Personne OPTIONAL,
     OU[6] IMPLICIT SEQUENCE OF
Unite-Organisationnelle OPTIONAL}

Liste-Attributs-Definis-Domaine ::= SEQUENCE OF Attributs-Definis-Domaine

Attributs-Definis-Domaine ::= SEQUENCE {
      type PrintableString,
      valeur PrintableString}

Adresse-X121 ::= NumericString

ID-Terminal ::= PrintableString

Nom-Organisation ::= PrintableString

Identificateur-Unique-UA ::= NumericString

Nom-Personne ::= SET{
 nom [0] IMPLICIT PrintableString,
 prenom [1] IMPLICIT PrintableString OPTIONAL,
 initiales [2] IMPLICIT PrintableString OPTIONAL,
 qualificateur-Genealogique [3] IMPLICIT PrintableString OPTIONAL}

Unite-Organisationnelle ::= PrintableString

Nom-Domaine-Prive ::= CHOICE{
      NumericString,
      PrintableString}

Types-Codage ::= [APPLICATION 5] IMPLICIT SET{
 [0] IMPLICIT BIT STRING {
     non-defini(0), tLX(1), texte-IA5(2),
     fax-G3(3), tIF0(4), tTX(5), videotex(6),
     voix(7), sFD(8), tIF1(9)},
 [1] IMPLICIT Params-Non-Essentiels-G3 OPTIONAL,
 [2] IMPLICIT  Params-Non-Essentiels-Teletex OPTIONAL,
 [3] IMPLICIT Capacites-Presentation OPTIONAL}

Params-Non-Essentiels-G3 ::= BIT STRING {
     bidimensionnel(8),
     haute-Definition(9),
     longueur-Illimitee(20),
     longueur-B4(21),
     largeur-A3(22),
     largeur-B4(23),
```

```
     non-Comprime(30)}

Params-Non-Essentiels-Teletex ::= SET {
jeux-Caracteres-Graphiques [0] IMPLICIT TeletexString OPTIONAL,
jeux-Caracteres-Commande [1] IMPLICIT TeletexString OPTIONAL,
formats-Page [2] IMPLICIT OCTET STRING OPTIONAL,
capacites-Diverses-Terminal [3] IMPLICIT TeletexString OPTIONAL,
usage-Prive [4] IMPLICIT OCTET STRING OPTIONAL}

Capacites-Presentation ::= OCTET STRING -- Pour ne pas importer T.73!!!


MPDU-Rapport-Remise ::= SEQUENCE{
    Enveloppe-Rapport-Remise, Contenu-Rapport-Remise}

Enveloppe-Rapport-Remise ::= SET {
     rapport Identificateur-MPDU,
     expediteur Nom-OR,
     Informations-Trace}


Contenu-Rapport-Remise ::= SET {
   initial Identificateur-MPDU,
   intermediaires Informations-Trace OPTIONAL,
   ID-UA-Contenu OPTIONAL,
   destinataires[0] IMPLICIT SEQUENCE OF
Info-Destinataire-Concerne,
   renvoye [1] IMPLICIT Contenu-UMPDU OPTIONAL,
   informations-Taxation [2] ANY OPTIONAL}

Info-Destinataire-Concerne ::= SET {
   destinataire [0]  IMPLICIT Nom-OR,
   [1] IMPLICIT Identificateur-Complementaire,
   [2] IMPLICIT Indicateur-Destinataire,
   [3] IMPLICIT Derniere-Informations-Trace,
   destinataire-Prevu [4] IMPLICIT Nom-OR OPTIONAL,
   [5] IMPLICIT Informations-Supplementaires OPTIONAL}

Derniere-Informations-Trace ::= SET{
arrivee [0] IMPLICIT Date-Heure,
convertis Types-Codage OPTIONAL,
[1] Rapport}

Rapport ::= CHOICE {
    [0] IMPLICIT Info-Remises,
    [1] IMPLICIT Info-Non-Remises}

Info-Remises ::= SET {
 remise [0] IMPLICIT Date-Heure,
 type-UA [1] IMPLICIT INTEGER
 {public(0), prive(1)} DEFAULT public}

Info-Non-Remises ::= SET {
     [0] IMPLICIT Code-Raison,
     [1] IMPLICIT Code-Diagnostic OPTIONAL}
```

```
Code-Raison ::= INTEGER {
incident-Transfert(0),
incapable-Transferer(1),
conversion-Non-Effectuee(2)}

Code-Diagnostic ::= INTEGER {
    nom-OR-Non-Reconnu(0),
    nom-OR-Ambigu(1),
    saturation-MTA(2),
    bouclage-Detecte(3),
    uA-Indisponible(4),
    delai-Max-Expire(5),
    types-Codage-Non-Acceptes(6),
    contenu-Trop-Long(7),
    conversion-Impossible(8),
    conversion-Interdite(9),
    conversion-Implicite-Non-Declaree(10),
    parametres-Non-Valides(11)}


Informations-Supplementaires ::= PrintableString



MPDU-Essai ::= Enveloppe-Essai

Enveloppe-Essai ::= SET {
    essai Identificateur-MPDU,
    expediteur Nom-OR,
    Type-Contenu,
    ID-UA-Contenu OPTIONAL,
    initiaux Types-Codage OPTIONAL,
    Informations-Trace,
    Indicateur-Message DEFAULT {},
    longueur-Contenu [0] IMPLICIT INTEGER OPTIONAL,
    bilateral[1] IMPLICIT SEQUENCE OF
Info-Bilaterales-Domaine OPTIONAL,
    destinataires[2] IMPLICIT SEQUENCE OF Info-Destinataire}
```

End of the definition of the MPDU type.